Department of Informatics

Assessment of genomic variant calling methods through simulations

Master thesis

Matias Holte

21st January 2013

Abstract

DNA sequencing has brought many important improvements to medicine, and is a field of active development. Through the study of genetic information, we gain knowledge of hereditary diseases and traits, both in humans and other species.

However, the process of sequencing is difficult, both due to vast amounts of data and sequencing errors. Verification of results is often done with expensive re-sequencing and analysis.

In this thesis we study the use of simulated reads in order to obtain exact results. First we suggest a method for creating a known artificial genome, using dbSNP to provide variation. Several existing programs for variant calling are evaluated through detailed analysis of variant files. Finally we suggest methods for improving verification of results.

The results show that the GATK variant callers performed well, but also Dindel provided some advantages. Furthermore, results suggest that some issues are caused by erroneous mapping and realignment.

We hope that others can use these results to improve the development of sequencing algorithms through simulations.

Acknowledgments

First of all I would like to thank my advisor, *Torbjørn Rognes*, for valuable discussions and excellent technical guidance. Your knowledge and experience helped me focus the efforts in the right direction.

I would also like to thank all the nice people and students in the 10th floor for motivation and good working environment, particularly through the weekly meetings led by *Geir Kjetil Sandve*.

The thesis was improved significantly by *Paul Kjetel Soldal Lillemoen*, who pointed out loads of weaknesses in the biological background, and *Inés Cendón Rodríguez*, who corrected countless spelling errors throughout the thesis. *Anette Johansen* and *Helga Holmestad* has also been very helpful with the correction.

Generally I am very happy for *Studentorchesteret Biørneblæs*, which provides a great social life at campus besides the studies, and also a concert after the thesis presentation.

Most of all I would like to thank my friends and family for your love and friendship, I am indeed lucky to have your support!

Motivation

The original topic of this thesis was to investigate the realignment algorithms in variant calling, and preferably try out new algorithms. But in the search for good verification methods, both for the algorithms yet to be implemented, but also for studying the existing alternatives, there were several issues that called for further study. We could not find any good and easy way of verifying the results from previous studies.

This frustration caused a change of focus from generating new algorithms to reviewing existing ones. In the process, some simple new tools had to be made, and these are described where appropriate.

There are many questions that can be asked, but in the end, it boils down to the simple question: How can one best evaluate the accuracy of an algorithm in bioinformatics?

Preferably, the evaluation should be feasible without resorting to expensive and error-prone re-sequencing.

Simulation is cheap, creates results that can be reproduced, and it is easy to obtain results with high precision. A major issue with simulations, however, is to obtain accurate, unbiased results. An important challenge is therefore to create simulated data that are as realistic as possible.

The second major difference with simulated reads is the verification process. With simulated reads, one have a golden truth of the entire genome, and the challenge is to ensure that comparison is done correctly.

Nevertheless, all steps in the pipeline can be analysed, not only read generation and variant verification. Thus there are many programs and algorithms that may be assessed through read simulation, and the availability of a golden truth allows even more detailed analysis of the results.

Outline

The theoretical background is described in the first four chapters. First chapter explains important biological background, whereas the second chapter focus more on the technological background of bioinformatics. Chapter three describes important software packages, before the problems, in which the software are utilised, are described in chapter four. This chapter also describe algorithms for solving the problems.

Chapter five describes the experiments, with commands, questions and hypotheses. Then chapter six presents the results from the experiments, and tries to answer questions from chapter five and comment interesting observations. These two chapters may be read in parallel, alternatively the result chapter can be read first, followed by the methods chapter if something is unclear.

A short discussion is provided in chapter seven, where the key findings are repeated. Also weaknesses with the experiments are described.

Finally there is a bibliography, an index of terms and an appendix with relevant code.

Contents

1 Genetics

This section provides background information on the biological part of bioinformatics. First we go through important terms and concepts in genetics and biology. This includes how the genome is built up and organized. Then we look at mutations in the genome, how they appear and why they are important.

1.1 Genetic information

The first part is a brief introduction to the biology in genetics. It is impossible to give a thorough background in a master thesis, and the reader is referred to a standard textbook on genetics, like [13, Chapter 1.2] for more details.

1.1.1 Basic structure

Proteins are among the main building blocks of our body, and play an essential role in many of the physiological processes. Simplified, the cells produce proteins by *transcribing DNA* into *RNA* and then *translating* RNA into proteins.

DNA, or deoxyribonucleic acid, consists of four *nucleotides*, namely *adenine* (A), *thymine* (T), *guanine* (G) and *cytosine* (C). These are attached to a sugar-phosphate backbone. Usually two DNA strings are paired, forming an anti-parallel double helix. Each nucleotide is paired with its *complementary* nucleotide. A is always paired with T, and C with G. Thus each of the two *strands* contains all the information in the other strand. Nucleotide pairs are also called *base pairs*.

RNA, or ribonucleic acid, has a slightly different backbone, and the nucleotide T is replaced by *Uracil (U)*. Furthermore, in most cases it consists of a single strand. There are several types of RNA, used for different purposes.

After the DNA has been transcribed into RNA, it is processed into mRNA, or messenger RNA, which can be transported into the cytoplasm of the cell where it is available to the *ribosomes* where the mRNA is read in groups of 3 consecutive nucleotides. These groups are called *codons*. Each tRNA, or transfer RNA, can bind to both a codon and an amino acid. Thus a sequence of codons can be translated into an amino acid chain, which can be further folded into the final protein.

In addition to the codons coding for one of the 20 amino acids, there are also special *stop codons* where translation ends. Because there are 4^3 codons, and only 20 amino acids, many codons translates into the same amino acid.

1.1.2 Organization of genome

The *genome* contains the hereditary material of the organism [13, Chapter 2.1], in humans it consists of DNA. The genome is organized in units called *chromosomes*. In humans there are three types of chromosomes: 22 autosomes, 2 sex chromosomes and one tiny, circular mitochondrial chromosome. The autosomes and sex chromosomes are stored in the *nucleus* of the cells, while each of the mitochondria contains a mitochondrial chromosome, inherited from the mother.

Ploidy denotes how many copies of a chromosome set can be found in the cells. Cells in healthy humans are either *haploid* or *diploid*. In haploid cells, only one copy of each of the 22 autosomes plus one sex chromosome are present, this is the case in *gametes* or sex cells. Diploid cells have 22 pairs of *homologous*, or similar, *autosomes*, plus a pair of two *sex chromosomes*. Females have two homologous sex chromosomes X, whereas

males have both an X and a Y chromosome which are not similar, the X chromosome being considerably larger.

Usually, each pair of chromosomes consists of one chromosome from each of the two gametes, one from each of the two parents. At some point the gametes merged and created a *zygote*; the initial cell that all other cells of an organism derives from.

Although we use the expression *chromosome pair* for homologous chromosomes, they are not physically paired in the nucleus. When chromosomes are pictured with an X shape, they are shown in the state after DNA replication, a part of *mitosis* or cell division. The point where the two copies meet is called the *centromere*.

The DNA is packaged very tightly in a *supercoiled* structure. First, DNA is wrapped around histone proteins to create *nucleosomes*. These are further coiled into chromatin fiber by connecting H1 histones. This structure is further coiled by attaching *scaffolds*. The combination of DNA molecules and binding histone proteines is called *chromatin*, what the chromosomes are made of.

Chromatin varies in density along the chromosomes. These variations can be visualized by applying a special dye. Dense chromatin, *heterochromatin*, is usually found around the *centromere* and the ends of the chromosomes, the *telomeres*.

The less dense chromatin is called *euchromatin*, and is where most genes are located. It is also more actively transcribed, and thus expressed. Heterochromatin, on the other hand, contains large parts of so-called *junk DNA*, parts that do not encode proteins and are seemingly useless. However, newer studies shows that these part of the genome may still have an important function [14].

1.1.3 Genome statistics

The human genome consists of slightly above than 3 Gbp (bp = base pairs, K,M and G are SI-prefixes). Between 200 and 250 Mbp are unsequenced heterochromatin. As humans are diploid, the total number of base pairs in a human individual is approximately 6 Gbp. The total number varies, both between sexes due to different length of the sex chromosomes X and Y, but also due to genetic variations.

The bases A and T are the most common, each constituting about 30% of all bases, whereas C and G constitute about 20% [27].

A *gene* is considered a unit in genetics and defined as a DNA segment that contributes to phenotype/function [41]. The total number of genes is unknown, but an estimate is about 20000 in the human genome. But both the number and definition of gene is unclear [34].

A *locus* (plural: loci) is a position in the genome, identified by a marker, that can be mapped. A gene may contain several loci, but a locus does not necessarily correspond to a gene, for instance in the case of non-coding regions.

1.1.4 Phenotype and genotype

The term *genotype* is used to denote the genetic information of the organism. In other words, the DNA inherited from its parents [13, Chapter 1.6]. Except for sudden mutations (for instance in cancer cells), it is constant for the entire lifetime of the organism.

The term is often used to mean *partial genotype*, where we restrict our attention to a small subset of genes we are interested in.

Phenotype is used to denote a set of traits of an individual. It can be harmless traits like hair color, or more serious traits like diabetes. Just as with genotype, we may mean *partial phenotype* when using the term phenotype.

The phenotype is influenced by the genotype, but also by environmental factors; for instance access to resources, temperature or diseases. Also the gene *expression* varies, not all parts of the DNA are transcribed and translated equally often. The field of *epigenetics* studies changes in gene function that cannot be explained by changes in DNA [2].

The phenotype may change continuously throughout the lifetime of the organism, in other words, fixed genotype does not imply fixed phenotype. But the reverse is also true, different genotypes can result in the same phenotype.

1.1.5 Alleles, zygosity and dominance

If a locus can have two or more forms, these are called *alleles* [13, Chapter 2.2,6.1]. Similar to genotypes, alleles refer to the genetic code. The difference is that alleles refers to variations in a single DNA strand, whereas the genotype is the product of both strands.

Assume there are two alleles in a particular locus. Without loss of generality, we may call them A and B. The three possible genotypes are then AA, AB and BB.

Because genes provide information for generating proteines, different alleles may code for different protein production, which again may result in different phenotypes. In the case of a *homozygote*, where both alleles are equal, the phenotype will follow the allele. In a *heterozygote*, the genotype is AB, that is, it contains two different alleles. If the phenotype shown by AB is the same as of AA, we say that A is *dominant* and B is *recessive*. This is because a single copy of A is adequate to obtain full expression, and the single copy of B is dysfunctional or insufficient to give expression. If AB neither resembles AA or BB, we either have *incomplete dominance*, where one copy only provides partial expression. Or we may have *codominance*, for instance in the case of blood type AB, where both alleles are expressed. A special case is when one copy is missing completely, a so called *hemizygote*.

1.1.6 Haplotype

Similar to alleles are *haplotypes*. They span over a segment of one chromosome, and are defined by a set of alleles. From the haplotypes of both chromosome copies, one can generate the genotype. The opposite is not true, however, as genotypes provide no information on how to combine heterozygous alleles.

In sequencing, haplotypes are important because all reads will be generated from one chromosome copy, in other words, belong to the same haplotype.

Furthermore, except for mutations, the haplotype of an organism stems entirely from one of the parents.

1.2 Mutations

The human genome is not static. DNA is vulnerable to changes from the environment and from imperfect replication. Changes in other cells may result in cell death or, in the worst case, cancer. [13, Chapter 15].

1.2.1 Point mutations and protein effects

Small mutations affecting only one or a few nucleotides are sometimes called *point mutations*. There are two main classes of point mutations, namely single-nucleotide polymorphisms (SNP) and indels.

*SNP*s (pronounced *snips*) are substitutions where one nucleotide is replaced by a different one. *Transitions* are when a nucleotide is changed within the same chemical category. These are *purines* (A and G) and *pyrimidines* (C and T). If a nucleotide is changed from a purine to a pyrimidine or vice versa, it is called a *transversion*. Each type of transition is about 2.8 more common than each transversion [6].

SNPs occurring at consecutive bases are sometimes denoted as *multi-nucleotide polymorphism* or *MNP*.

Due to grouping into codons, the effect on a SNP depends on its context. If the SNP changes an AGA codon into an AGC codon, the specified amino acid, argine, remains the same. This is called a *synonymous mutation*, and has no effect on the phenotype. Another name is *silent mutation*. However, if the codon is changed to specify a different amino acid, we have a *missense mutation*; for instance if the SNP changes AGA to AAA, which codes for lysine. Some amino acids are chemically similar to each other, and replacing one with the other may have only minor or no effects on protein functionality. Such missense mutations are called *conservative*. Similar, if the alternative amino acid is dissimilar, the mutation is called *nonconservative* and is more likely to cause adverse effects. Finally, if the codon is changed to a premature stop codon, for instance if AAG is changed to TAG, we have what we call a *nonsense mutation*. The generated protein will be shorter than the original, often with loss of function, depending on where in the chain the stop codon was introduced.

Indels (the term is a composition of *in*sertion and *del*etion) are mutations where one or more base pairs are either inserted or deleted. Insertion or deletion of a number of base pairs not divisible by 3 leads to a *frameshift mutation*, where all subsequent codons will start at a different position. The rest of the amino acid chain bears little resemblance to the original, both in length and content, and the resulting protein is likely to be unusable.

1.2.2 Mutations in non-coding regions

Mutations can also happen in regions which do not code for proteins, and even though the protein structure is unaffected, the expression may change. This is because transcription requires special binding sites, where special proteins can attach and start the transcription process. Predicting the effect of such mutations is harder than for mutations in coding regions.

1.2.3 Causes of mutations

There are several processes at molecular level that lead to mutations. There are both *spontaneous* mutations, happening frequently through random processes, and *induced* mutations, caused by the environment.

Spontaneous substitutions, usually in the form of transitions due to the chemical similarity, have various causes. It can be due to naturally occurring *isomers*, molecules that are similar to the normal nucleotides, but can form wrong bonds. The same error may also occur if the nucleotide is ionized. Nucleotides may also degenerate to other molecules which cannot specify the complementary pair uniquely. This is the case in *depurination* and *deamination*.

Spontaneous indels can be formed during DNA synthesis, when the template and new strand slip apart, and are rejoined at different positions. This is typically the case with repetitive areas, where parts of the new strand can match at multiple sites.

Mutations may also be induced, either intentionally in laboratories, or by accident by environmental factors. Examples include UV-light, ionizing radiation and various molecules. These can damage DNA in similar ways as by spontaneous mutations, but also more serious structural damage like strand breaks.

1.2.4 DNA repair mechanisms

Unlike other molecules in the body, DNA is actively repaired, rather than replaced. Many DNA damages and the corresponding repair mechanisms are described in [4, 12]. Some damages are repaired perfectly, but other repair mechanisms are error-prone, and may generate mutations, in particular during mitosis. Different organisms have different repair mechanisms.

The easiest errors to recover is when a damaged site can be uniquely reverted to the original nucleotide. This is, for instance the case for various methylations of nucleotides. The mutation can be repaired error-free, even without a template strand.

A more general approach is either *base excision repair (BER)* or, for more serious damages, *nucleotide excision repair (NER)*. When an error is discovered, enzymes and proteins are attached to the double helix. First the damaged part is removed, before the missing nucleotides are replaced by DNA polymerase.

In the case of mismatches, there is no damage to the nucleotides themselves, but the pairing is wrong. During *mismatch repair*, specialized proteins attach to the site, and look for a special methylation signature in order to identify the newly created strand, which is then excised away and regenerated.

Correction mechanisms exist when both strands are damaged, but these are error-prone and may lead to deletions or rearrangements in the genome. An example is the *nonhomologous end joining (NHEJ)*, which reattaches the broken strands.

Buildup of mutations can cause loss of function, which eventually cause a state of cell death or starvation. In this case, there are no daughter cells which would inherit the mutations. However, if the cell regulating mechanisms are damaged, the cell may enter a stage of uncontrolled replication, or cancer.

1.2.5 Large scale mutations

When repair mechanisms fail to reattach broken strands correctly, large parts of DNA may be rearranged. This is the case with *inversions*, where a string of DNA is reversed, and *translocations*, where DNA is moved to another position on a different chromosome.

In *copy number variations*, the length of the genome is altered, either through insertions or deletions. It is distinguished from indels by the size of the altered region, which is typically larger than 1 kb [44, 11].

1.3 Inherited mutations

In addition to the spontaneous mutations described in section 1.2, mutations in gametes can be passed on to the genome of the child. Mutations are the reason why we exist, and through evolution we have become the species we are today [13, Chapter 17-19]. The process consists both of selection and random genetic drift.

1.3.1 Selection

Mutations cause new alleles to be created. Assume that there are two alleles, B and b at a particular locus. Each allele is equally likely to be passed on to a gamete and to

the next generation. Thus, if the alleles are equivalent, the frequency of each allele is expected to be the same. On the other hand, *fitness* may skew the balance towards one of the alleles. Fitness gives the expected number of offspring and is given both by the probability of surviving until mating age, and the reproduction capabilities.

If the genotype BB has higher fitness than bb, with Bb being somewhere in the middle, more copies of the B allele will be passed on, and eventually dominate in the population.

In some cases, the heterozygote Bb may be the most fit, then the frequency will converge towards an equilibrium.

Fitness may be influenced by the genotype of the population as a whole, then it is said to be *frequency dependent*.

1.3.2 Genetic drift

While selection will drive the allele frequency towards the equilibrium, genetic drift will add a random factor. This is because genetic inheritance is a stochastic process with a finite population and thus a non-zero standard deviation. This is a particularly important factor when a small group of people have a large number of descendants, for instance after emigration.

Random mutations will also cause a slow drift away from the most frequent allele. If the selection process is slow because the fitness is nearly identical, these mutations may skew the equilibrium closer towards the middle.

2 Sequencing, assembly, mapping and variant calling

This section will cover background information on how the process of sequencing is done. That is, the informatics part in bioinformatics. Both important concepts and file formats are discussed. We also give a glimpse of sequencing machines and their characteristics. Finally we give a short overview over important areas and terms in statistics.

2.1 Introduction

By *sequencing* we mean the process of determining the nucleotide order of a physical DNA fragment using a sequencing machine. The sequences are called *reads*, and are organized into larger *contigs* during *assembly*. The motivation is typically either to find the DNA sequence of a species, like in the *human genome project*. This is called *de novo assembly*[33]. The alternative, to sequence an individual given prior knowledge of the genome of the species, is what we do in this thesis.

2.2 Sequencing technologies

For over 30 years, *sanger sequencing* was the major technology for DNA sequencing. It is characterized by relatively long reads (400-900 bp)[28], with very high accuracy, but very low throughput. Long reads are particularly good for de novo assembly, and the technology is mature, but due to the low throughput, the technology is slow and expensive.

The last 10-20 years, we have seen the development of *next generation sequencing* (NGS, also called *high-throughput sequencing*), which use massive parallelism to obtain huge number of reads at a low cost. Nevertheless, the reads themselves are shorter, which makes it computationally harder to align them. This, along with the sheer amount of data, which can amount to billions of base pairs each day, has inspired the developments of new and improved algorithms.

Because reads from NGS platforms are short and error-prone and the genome contains repeats, alignment of reads may be ambigous. This makes de novo assembly difficult. However, most of the sequencing technologies return mate pair information [8], in which two reads are paired with an approximate distance in the genome. This is particularly useful if one of the reads cannot be placed deterministically.

To get an impression of characteristics of reads from modern machines, we can look at some major NGS platforms [28]. These are *454 GS FLX* from Roche, *SOLiDv4* from Applied Biosystems and *HiSeq 2000* from Illumina. Whereas 454 have long read lengths of 700 bp, the other have reads from 50-101 bp. The accuracy of HiSeq is about 98%, which is lower than the other sequencers. The run time varies from 24 hours to 14 days, with output ranging from 0.7 Gb to 600 Gb. The price is also very different. HiSeq costs \$0.07/Mb, while 454 costs \$10/Mb. For comparison, Sanger sequencing costs \$2400/Mb

In other words, it is essential to choose the correct technology depending on the needs, regarding read length, accuracy, cost and throughput.

2.3 Mapping

The *reference* sequence is a consensus sequence generated from a sample of individuals, in our case humans, through de novo assembly. Short overlapping reads are merged to-

gether to longer *contigs*, which then are joined together during *scaffolding*. The resulting reference has a close similarity to other individuals can be seen in section 2.4.4.

Given that a reference has already been found, we may use it to find candidate positions of the reads, drastically reducing the number of potential overlaps. This way of aligning reads to the reference is called *mapping*. Because of mutations and sequencing errors, the task is still nontrivial, but for all reads that are near identical to the reference, it is very fast to look up all matching positions.

Mapping has a different advantage too; because of the close similarity to the reference, we can encode the assembled genome using an efficient variant representation, where only the differences from the reference are described.

2.4 What are variants?

Where two or more alleles exist, we have *genetic variation*. *Variants* are differences from the reference genome that occur in one or more individuals or individual cells.

Finding variants is the main goal in assembly with reference, and is called *variant discovery* or *variant calling*. Traditionally it is done mostly on humans, although population genomics of other species, like for instance fish [31], has gained increased interest the latest years.

Through variant calling we find what characterizes an individual genetically. Knowledge of variants may allow better treatment, aimed at an individual level, also known as *personal medicine*, or at least treatment on a *haplotype* level [38].

Of particular interest is the study of cancer cells, in which the DNA is altered compared with the other cells of the body. Sequencing cancer cells may reveal particular weaknesses which can be targeted by specialized medicine.

Variants stem from mutations, almost all of which are inherited from parents. They can be found in different areas, both in protein-coding regions, where they may change the amino acids in the protein, and in *noncoding DNA* inside or in intergenetic regions, where the *expression* of a gene may be altered. On average, humans have about 250-300 variations with loss of function [7].

There are several different ways of encoding variants. These are not unique, but most programs will try to find the easiest representation of the differences in the genome. The frequencies of variants is also described in section 2.4.4.

2.4.1 SNPs

Single-nucleotide polymorphism (*SNP*, pronounced *snip*) is a change in a single nucleotide at a given position compared to the reference genome. If you align the gene to the reference, only one position will differ. Almost all SNPs have only two *alleles*, or variants, in the whole population, and the SNP frequency is about 0.05-0.1% in humans. [37]

2.4.2 Indels

Short insertions or deletions are referred to as *indels*. If change in a coding region is not divisible by 3, we get a *frameshift mutation*, which affects all the subsequent codons. Even though indels typically refer to variants shorter than 1 kb [11], the mean deletion and insertion is only 5 bp and 8 bp respectively. The deletion frequency per site is about 0.44%, and the insertion frequency is about 0.16% [32].

2.4.3 Structural variations

Structural variations (SV) is a common term used for many different large ($\geq\sim$ 1kb) variations in the genome, both those changing the length, such as insertions and deletions, and those which do not, such as inversions and translocations [44, 11].

Large insertions and deletions, are often referred to as *copy number variations* (CNV) and can have a *dosage effect* in some genes, a change in the amount of a protein that is produced.

2.4.4 Variant frequencies

When talking about variant frequencies, it is crucial to be aware of how the reference is built. Siblings will typically have much less variation than two completely unrelated persons. Results can also be inaccurate when the samples are taken from a small geographical area, like in the case of the GRCh37 reference which was built from a few volunteers from New York.

In addition to the reference genome, we also have variant databases, where known alleles are added. As the databases increases in size, so does the probability of finding particular variants in a new sample. This is, however, correlated with the background of the sequenced individuals. Isolated native tribes could possess a larger amount of unknown variants.

A crude estimate on the number of SNPs in one individual is 3-3.5 million. In the human genome as a population as a whole, the estimate is between 10 and 24 million SNPs [27]. Compared to the number of humans today, this is a low number, but on the other hand, all humans descend from a small population, and new mutations are introduced slowly.

A plausible example can be found one of the studies that has sequenced a human individual [21]. 4,118,889 variants were found, of which 1,762,541 were heterozygous SNPs, 1,450,860 were homozygous SNPs, 38,985 were heterozygous MNPs, 14,838 were homozygous MNPs, 263,923 were heterozygous indels, 275,512 were homozygous insertions, 283,961 were homozygous deletions, 28179 were complex and there were 90 inversions.

An interesting observation is that indels are much more likely to be homozygous than SNPs and MNPs. This can be explained by selection, as random indels are more likely to be deleterious due to frameshift mutation.

In 2007, at the point of the study, 1,288,319 of the variants were novel or unknown to the variant database dbSNP. With the vast increase in known variants the last years, this number would have been reduced if the study had been carried out today.

As previously stated, this is just a summary of a single human individual. Furthermore, errors in variant calling and sequencing can cause both false negatives and positives. These numbers should only be used as a crude estimate of the true distribution.

2.5 Variant databases

Variant databases are ways of expressing the genetic variation, as a supplement to the reference genome. They can both be used as a source of genetic studies and to reduce the number of sequencing errors.

2.5.1 1000 Genomes Project

The *1000 Genomes Project* (1000G) [7] aims to systematically sequence the complete genome of at least 1000 humans across the world. The data can be used freely by scientists through public databases, and are an important source for studying genetic variation. Parts of the material study inheritance, as both a child and its parents are sequenced.

The size of 1000G is relatively small, but in return, the quality of the data is good. Each variant is coupled with the alleles in each individual. This way it is possible to find an estimate of the frequency of each allele in the population. Because entire genomes are sequenced, it is also possible to study the correlation between different variants.

2.5.2 dbSNP

dbSNP [17] is a public database of genetic polymorphisms. Despite its name, not only SNPs are in the collection, but also indels and other genetic variation submitted from labs across the world. Even though the database have improved SNP discovery and detection, the data are not always accurate [30]. False positives are much more likely to be submitted than in the case of 1000G. There is also the problem that not all indels in dbSNP follow the standard of left-aligning (see section 4.7).

Version 135 of dbSNP have listed 52,716,087 simple variants. Of these, 84.276 % are SNPs, 8.056% deletions and 7.264% insertions. The remaining 0.40% are MNPs. The most common are where 2 nucleotides are replaced, with a frequency of 0.200% and 3 nucleotides with a frequency of 0.175%. Longer MNPs, between 4 and 160 nucleotides, are infrequent, adding up to less than 0.03%.

2.6 File formats

Over the past years, a large amount of file formats have been created. While originally developed to store data from a specific program, some of these formats have now become a *de facto* standard for that particular file type. Such standardization benefits downstream analyses, which can use generic algorithms suitable for multiple platforms.

The file formats most relevant to variant discovery are described in this section. From each format, an extract of a typical file is provided, with ellipsis denoting skipped lines.

2.6.1 Sequence data

The *FASTA* format is used for describing either nucleotide or peptide sequences. Each letter in the file describes exactly one nucleotide or peptide position, or a gap of indeterminate length. It is also possible to describe any combination of 2,3 or 4 nucleotides, in case the position is uncertain. The most common is *N*, which represents *aNy* nucleotide.

Each sequence in the FASTA file begins with a description line, followed by non-empty lines of sequence data. The description line (defline) starts with the greater-than symbol (>), whereas the sequence code is defined by the IUPAC/IUB nomenclature.

FASTA files are commonly used for reference sequences. It usually comes with the *.fasta* or *.fa* file extension.

Example

From human_g1k_v37.fasta, the reference genome from 1000 genomes.

```
>1 dna:chromosome chromosome:GRCh37:1:1:249250621:1
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
...
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNTAACCCTAACCCTAACCCTA
ACCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCTA
ACCCTAACCCTAACCCTAACCCTAACCCAACCCTAACCCTAACCCTAACCCTAACCCTAA
...
N
>2 dna:chromosome chromosome:GRCh37:2:1:243199373:1
...
```

2.6.2 Sequence and quality data

The *FASTQ* file format, originally a standard published by Sanger, is used for bundling together sequence data and quality scores. It extends the FASTA format by attaching a numeric score (encoded as an ASCII character) to each nucleotide in the FASTA sequence [5]. The FASTQ format has become a *de facto* standard for storing unaligned *raw reads*. Typical file extensions are *.fastq* or *.fq*.

Quality scores are used to describe the probability of an incorrectly called base. There are 3 different incompatible FASTQ formats, which differ in offset, range and quality score type. These are the Sanger, Solexa and Illumina FASTQ formats. Both quality score types used are logarithmic, the *Phred* scale used by Sanger and Illumina is on the form $Q = -10log_{10}p$ where Q is the score and p is the probability of error. All scores can, however, be converted to each other, possibly with rounding issues.

Another important format is the *sra* format used by the *sequence read archive (SRA)* (section 2.7.1). Such reads may be converted to FASTQ or others by using the SRA toolkit.

Example

From simreads.A_fir.fastq, as generated by GemSim. Line breaks were added for readability, this is indicated by indenting next line.

```
@r1_from_1_ln396_#0/1
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
        NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
+
hhfefhdhhdghhhhfghhffhgfhhcghff_ghgfhhgNhhLfhg‘g[hhhheh
        beachgghda‘BdbhhgcfehPBdgf]g][cgZeBdd_aadWgcB
@r2_from_1_ln358_#0/1
AAAAAAAAAAAAAAAAAAAAAAAAAAAAGCCGGGTGCGGGGGCTCACACCTGTAAT
        CCCAGCACTTTGGGAGGATGAAGCGGGCAGATCACTTGAGGTTGG
+
hdhahhhhhhhahhfhhgfhhhdhghhhhhhhghhghGhPhfdhh\efdhhghg
        gagafBchhgadaghheead_RaaabdcacBBaf‘aBB]aB\‘aB
@r3_from_1_ln340_#0/1
GGTTTCAGTTCCAGCAACTTTGAATGAAAAATCCGTAGTTCACAACATTCTGAGAC
        ATGTTTATTCCTTTATTCATTTGCTCACCAGCTGTTTCCTGGAGA
+
```

```
hgfhhhhhg\ghhhhgfdchehcf‘hghdhahhhhhhfhhhhhhfgdhfhhhheeB
        hffhehhhd‘dfdg_afdg^‘eha_hb[c‘BddedbaUa_Yc\^[
```

2.6.3 Aligned sequence data

Sequence data aligned to a reference sequence are typically given in a *Sequence Alignment/MAP* (*SAM*) file [25]. Both short and long reads, up to 128 Mbp, are supported. The *Binary Alignment/Map* (*BAM*) file format is the binary equivalent file format to SAM, and contains the exact same information, but compressed to reduce file size and consequently costly disk reads. The compression library used, BGZF, is developed to allow fast random access in the compressed file. Together with sorting by coordinate and indexing, this allows fast retrieval of alignments in a particular region.

SAMtools (section 3.5) is a software package designed to parse, manipulate and write aligned sequence data in the BAM/SAM file format.

Sequence data can be converted by samtools to the *pileup* format. This format is useful to visually inspect all reads at a given position in the reference, almost like a text-based version of the program *Interactive Genomics Viewer*. This format is also used for some variant calling algorithms in VarScan and samtools.

Example

As BAM files are binary, only the uncompressed SAM format can be shown, here are lines from BWA. The file is tab-delimited, and for readability, line feeds are added.

```
r8769436_from_1_ln387_#0        99      1       60001   29
     11S90M  =       60276   376
     NNNNNNNNNNNGATCCAGAGGTGGAAGAGGAAGGAAGCTTGGAACCCTATAGAG
         TTGCTGAGTGCCAGGACCAGATTCTGGCCCTAAACAGGTGGTAAGGA
     ghahhfXfhhhhhhfhhhYhhfhhhhahhhchhfhhh‘fhh^hhce\fhdhghc
         hghhhchfdhhgddgbB]gcfhEhagaBaB\B\c]ahaBeBaaaWWa
     MD:Z:65C24      RG:Z:1  XG:i:0  AM:i:29 NM:i:1  SM:i:29
     XM:i:1  XO:i:0  XT:A:M
r315821_from_1_ln349_#0 89      1       60005   37
     101M    =       60005   0
     CAGAGGTGGAAGAGGAAGGAAGCTTGGAACCCTATAGAGTTGCTGAGTGCCAGG
         ACCAGATCCTGGCCCTAAACAGGTGGTAAGGAAGGAGAGAGTGAAGG
     [ad\aV_ah‘cg^]d[ada‘\eaaBadhdhYghbfdhdf]dcfchhheefhhch
         dhfgghhgcghhhgggghhhhehehghhhfhhhhghhhhhhhhhghgh
     X0:i:1  X1:i:0  MD:Z:101        RG:Z:1  XG:i:0  AM:i:0
     NM:i:0  SM:i:37 XM:i:0  XO:i:0  XT:A:U
```

2.6.4 Variant storage

The *Variant Call format* (*VCF*) [9] is a file format developed for the 1000 genomes project [7] to store variant data like SNPs, insertions, deletions and structural variations.

Each data line in the VCF file contains information about a variant at a position in the genome. It stores both the actual variant and various meta data, like quality score, sample statistics and database memberships.

VCF files are usually compressed with the same BGZF library as SAM files, creating .gz-files. An alternative compression method is to write it in the binary BCF format (section 3.5).

Example

This VCF file from GATK HaplotypeCaller has a large header, therefore many lines are omitted. The header lines are required if the fields are present in following data section, typically in the INFO column. The fields are tab-delimited. For readability, white space is modified in the example.

```
##fileformat=VCFv4.1
##FILTER=<ID=LowQual,Description="Low quality">
##FORMAT=<ID=AD,Number=.,Type=Integer,Description="Allelic
    depths for the ref and alt alleles in the order listed">
...
##INFO=<ID=extType,Number=1,Type=String,Description="Extended
    type of event: SNP, MNP, INDEL, or COMPLEX">
##contig=<ID=1,length=63025520,assembly=b37>
...
#CHROM   POS   ID   REF   ALT   QUAL   FILTER   INFO   FORMAT   1
1        62255  .   T     C     1733.77  .       AC=1;AF=0.500;AN=2;
    ActiveRegionSize=197;BaseQRankSum=0.168;ClippingRankSum=1.846;
      DP=52;EVENTLENGTH=0;FS=0.000;HaplotypeScore=46.6733;MLEAC=1;
      MLEAF=0.500;MQ=60.47;MQ0=0;MQRankSum=0.504;NVH=2;
      NumHapAssembly=6;NumHapEval=3;QD=33.34;QDE=16.67;
      ReadPosRankSum=-0.932;TYPE=SNP;extType=SNP
    GT:AD:GQ:PL    0/1:21,31:99:1762,0,1145
...
1       614256  .   T     G     19.81   LowQual   AC=1;AF=0.500;AN=2;
    ActiveRegionSize=161;BaseQRankSum=-0.370;ClippingRankSum=-1.719;
      DP=47;EVENTLENGTH=0;FS=5.820;HaplotypeScore=47.8527;MLEAC=1;
      MLEAF=0.500;MQ=58.80;MQ0=0;MQRankSum=0.979;NVH=1;
      NumHapAssembly=5;NumHapEval=3;QD=0.42;QDE=0.42;
      ReadPosRankSum=0.675;TYPE=SNP;extType=SNP
    GT:AD:GQ:PL     0/1:44,3:48:48,0,2482
...
1       126160  .   C     CAAA  9691.77  .       AC=2;AF=1.00;AN=2;
    ActiveRegionSize=258;DP=46;EVENTLENGTH=3;FS=0.000;
      HaplotypeScore=33.0556;MLEAC=2;MLEAF=1.00;MQ=52.79;MQ0=0;
      NVH=4;NumHapAssembly=16;NumHapEval=3;QD=210.69;QDE=52.67;
      TYPE=INDEL;extType=INDEL
    GT:AD:GQ:PL     1/1:0,46:99:9720,1082,0
```

2.6.5 Index files

To allow fast random access to the potentially large files, they can be sorted and indexed. These indexes are usually put in separate *index* files with the same name as the original file, but with an additional file extension. Both VCF, BAM and FASTA files can be indexed. VCF files are given an *.idx* extension, whereas tabix-indexed gz-compressed VCF files are given an *.tbi* extension. BAM files are given an *.bai* extension, and the original .bam extension may be omitted. FASTA files are given an *.fai* extension.

If index files are required for an algorithm, it will issue a warning and either abort or generate the index file(s) automatically

2.6.6 Nonstandard file formats

In addition to the standard (or de-facto standard) files, many programs create additional files as they see fit; for instance dindel (section 3.2) creates many intermediate files.

2.7 Data sources

In order to test algorithms for mapping and variant calling, it is essential to have input data, either raw sequence data in FASTQ format, or aligned data files in SAM/BAM format. Such sequence data can be generated in two ways.

2.7.1 Biological samples

The typical use of sequencing technology is to find the true DNA sequence from a given biological sample. For this purpose, we may extract a sample and use sequencing platforms from companies like Illumina or Roche to obtain raw reads.

To simplify data analysis, samples from individuals can be downloaded from genetic sequence databases like the *sequence read archive (SRA)* [20, 42] run by the *National Center for Biotechnology Information (NCBI)*. As of September 2012, this archive contains more than 300 Terabases freely available for download.

Both the *Illumina*, *SOLiD* and *Roche/454* platforms are represented, with Illumina being the dominating technology.

Though human sequences dominate the collection, with more than 60% of the bases, other species like *Mus musculus* are also available.

In addition to SRA, several programs also include read samples. GATK, for instance, provides a package with aligned sequences from chromosome 20 from the human individual with code NA12878.

An important disadvantage with biological samples, is that the true gene sequence is unknown, and it is impossible to know the exact number of correctly called variants. This disadvantage can be partially negated in experiments by expensive and inefficient Sanger re-sequencing. Due to the high cost, this is usually only done on a small sample.

2.7.2 Artificial samples

An alternative to biological samples is to *simulate* reads from a given reference sequence. A program is used to generate plausible reads, which could have been generated by sequencing platforms, if the biological sample had contained that particular sequence.

The clear advantage is that there exists a golden truth, the reference sequence. This way it is possible to test the correctness of mapping and variant calling algorithms.

On the other hand, we risk testing on unrealistic environments, in particular if the parameters for simulating reads and for downstream analyses are matched; for instance some simulators set max indel size to a constant.

The generation step is not trivial. Not only should it simulate the error characteristics of the sequencing technology, but also take known variations into account. In other words, it should introduce both random read artifacts and true variants to be detected later.

The generation process is usually done in two steps. First, variants such as indels and SNPs are introduced into the reference genome, before reads are drawn from the modified reference.

As humans are *diploid*, we will typically see many variants occurring in only one copy of the chromosome pair. This conflicts with the idea of using a single reference sequence

as a source. Furthermore, these variants are mostly inherited and thus shared among multiple individuals in the population. Said another way, variants are not independent, and the simulator should take that into account.

That different cells may have different DNA (e.g. in cancer cells) is outside the scope of this thesis.

2.8 Statistics and metrics

Another important topic is statistics. Not only is it important to find the correctness of the different algorithms, but also which kinds of errors occur.

2.8.1 True/false positives/negatives

Assume that we would like to estimate which genome positions have a certain variant. Each position may either have the variant or not. And we may either estimate that the variant exists or not.

A *true positive* is a positive estimation that is correct. We believe the variant is present, and when we check with the data, that estimation is true. A *false positive* on the other hand, is an incorrect positive estimation. We believe the variant is present, but in reality it is not. Similar is the *false negative*, where we estimate that the variant is not present, but it really exists. Finally a *true negative* is when we correctly believe that a variant is not present.

Positive/negative always refers to the experiment result, whereas true/false tells us whether the result is correct or not.

Throughout the thesis we may use TP,FP, FN and TN to denote true positives, false positives, false negatives and true negatives.

2.8.2 Sensitivity, specificity and precision

From true/false positives/negatives, we may calculate multiple metrics, also used in information retrieval.

Sensitivity, also called *recall*, is defined as $sensitivity = \frac{TP}{TP+FN}$, the probability of a positive estimation, given that the mutation actually exists.

If the number of mutations, TP+FN, is constant, the use of sensitivity and TP is equivalent.

Specificity is the opposite, $specificity = \frac{TN}{TN+FP}$, the probability of a negative estimation, given that the mutation does not exist.

Because the number of true negatives may be very high, like if the probability of a mutation is very low, specificity can be very close to 100%. In this case it may be more helpful to use *precision*, defined as $precision = \frac{TP}{TP+FP}$, or the probability of a positive estimation being correct.

Sensitivity and precision are connected; it is hard to improve one without sacrificing the other. Sensitivity may be increased by using a more aggressive algorithm which marks many positions as positive, even with weak evidence. This will typically cause more FP and thus a lower precision.

Using these metrics, it is possible to compare several algorithms, by adjusting aggressiveness parameters until the sensitivity is the same, then compare precision for several sensitivity values.

Receiver operating characteristic is used in many cases, but due to the high specificity, a so called ROC curve with specificity and sensitivity along the axes would not

provide a good image when the number of TN is high. The specificity would be close to 100% for most sensitivity levels.

To convert from sensitivity and precision to TP, FP and FN, one needs to know the total number of variants (TP+FN). Then the values are calculated as follows:

$$TP = sensitivity * total$$

$$FN = total - TP$$

$$FP = \frac{TP}{precision} - TP$$

2.8.3 Poisson distribution

Assume we have a large number N of independent events, and that each event may happen in an interval with a certain probability p. The mean, or the expected number of successes is Np. A success is defined as an event happening in the interval. But the actual number of successes will be an integer that varies around the mean, in a *binomial* distribution. If N is large and p is small, the distribution approaches the *Poisson* distribution, defined as

$$f(k;\lambda) = Pr(X = k) = \frac{\lambda^k e^{-\lambda}}{k!}$$

The advantage of Poisson is that it only depends on the mean, $\lambda = Np$, not the sample size.

3 Software packages

In this section we will first briefly describe the most important program packages, with licence, reference to papers or homepage and other noteworthy information. The different steps in the variant calling pipeline will be presented in more detail in section 4, where the relevant tools in the different packages are described and theoretic differences discussed.

3.1 GATK

The *Genome Analysis Toolkit (GATK)* [10] is an almost complete framework for the whole pipeline, covering all steps from mapped reads to variants ready for analysis. The paper discusses current strategies and issues with variant calling, and compares them with this new framework in the GATK. Where some steps in the pipeline in figure 1 on the following page are missing in GATK, good alternatives already exist.

A clear advantage with the pipeline structure is the ease of replacing a single utility or algorithm. During the writing of this thesis, an experimental variant caller was published, and this can easily be exchanged with the stable and well-tested algorithm.

It is actively developed at the Broad Institute, with licences covering both for-profits and universities (open source). It is run as a java program, but with a parameter which specifies which utility or *walker* to use.

A separate program, *GATK-Queue* has been developed to manage the numerous tools and data in a typical pipeline. It can also split input files to take advantage of parallel processing capabilities. Work flow is described by a so-called *Qscript*. Queue was considered, but not used in this thesis.

3.2 Dindel

Dindel [1] is a program which focuses on discovering indel variations in short reads, rather than SNPs or structural variations, using a bayesian approach. Both a realignment and a variant calling algorithm are included.

It requires BAM files with read-alignments with well calibrated mapping qualities, like from Stampy, BWA or MAQ. Currently (version 1.01) it is only appropriate for Illumina data, not 454 data.

The main program is written in C++ and supported and tested on Linux and Mac. There are also some tools written in python.

It is developed by Albers, Lunter and Durbin at the Wellcome Trust Sanger Institute and at the University of Oxford, and source is freely available under the GPL licence.

3.3 BWA

The *Burrows-Wheeler Aligner* [23, 24] is a program for doing gapped alignment of reads, both paired-end and single-end reads. Its algorithm is described in section 4.2. In addition, it provides a utility for indexing FASTA files. BWA is widely referred to by other programs. C source code and program is available with a GPL licence.

3.4 SRMA

Short Read Micro-reAligner *(SRMA)* [16] is an algorithm for re-aligning mapping results from an existing aligner. It tries to combine the results from the mapper to improve

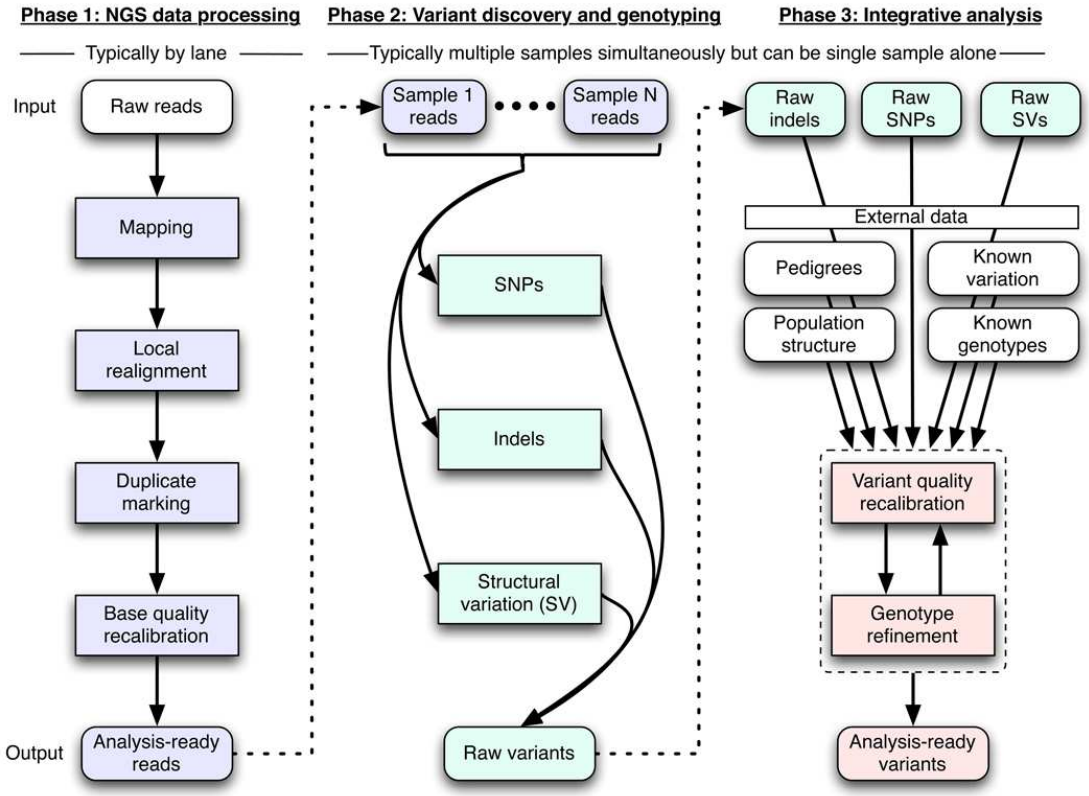

Figure 1: GATK pipeline. Copied from [10].

the final alignment. As the name implies, only a very small window is considered for each read. The default is 20 bp in each direction.

From a pipeline perspective, SRMA can be seen as an add-on to existing aligners, extending the aligning step by additional algorithms.

Both java and C source are available under the GPL licence. However, neither the main web page, nor the source code, have been updated since march 2011, with version 0.1.15 being the last one.

3.5 Samtools and bcftools

As described in section 2.6.3, samtools [25, 22] is a program used for manipulating aligned reads in the SAM/BAM format. It is particularly useful for converting between the different formats, but it can also sort, index and extract parts of the alignments.

Samtools is open source software hosted by SourceForge. The program is written in C for linux, though a windows port with limited functionality exist. Input/output can be given to standard in/out, to facilitate the use of a standard unix pipe. File output is possible through stdout redirection.

Hosted on the same page are also some related utilities. *bgzip* is used for compression/decompression and *tabix* is used for indexing of various file types. *bcftools* can convert between compressed BCF files and uncompressed VCF files and index BCF files.

Samtools also contain a variant calling algorithm which works in two steps. First by running the samtools mpileup command, which generates a pileup file. Then the bcftools view command can call variants using bayesian inference. There also exist a

samtools pileup command, which has been deprecated in favor of mpileup for generating pileup files.

3.5.1 Picard tools

Picard has its own project page [43], but is supported through the samtools mailing list. It provides many utilities related to SAM/BAM files and complements Samtools in many ways. In this thesis it was used to modify alignment files to conform to restrictions in GATK. The utilities are written in java and source code is available.

3.6 VarScan

VarScan [18, 19], also called VarScan 2 from the version number, is a program for detecting variants. The second article refers to an experiment where reads from tumor cells were compared to reads from healthy cells to find cancer variants.

VarScan 2 takes only (m)pileup files as input, so these must be generated by, for instance, samtools. Output is also non-standard, instead of a vcf file, VarScan outputs a tab-delimited file with a special set of columns. The first of these columns are chromosome name, position, reference allele and consensus genotype of sample. But there is an *–output-vcf* option which cause varscan to output in VCF format as well.

The variant calling programs mpileup2snp and mpileup2indel calls only SNPs and indels respectively.

VarScan is provided as a multi-platform java application, and source code is available. It is developed by Daniel Koboldt and others at Washington University.

3.7 VCFtools

VCFtools [9] is a set of small programs, either written in perl or C++. It can read, parse and modify VCF files in multiple ways, for instance validating, merging, intersecting and extracting statistics on variant files. Some of these tools are described in section 4.6.1.

VCFtools is provided as an open source package through SourceForge.

3.8 IGV

The *Integrative Genomics Viewer* [40] is a graphical program for manually inspecting reads and reference genome. While the program does not do any variant calling or aligning itself, the visualization tools make it easier to get a quick glance at what characterizes the difficult regions that the other programs struggle with. Thus one may quickly come up with good hypothesis which can be tested by normal means.

IGV is provided from the Broad Institute as an open source java program, freely available to multiple platforms. Other visualization programs exist as well, but as this is not the main topic of this thesis, those have not been considered.

4 Software solutions

In this section we will go through all steps of the pipeline and describe tools that can be used, with theoretic advantages and shortcomings of the various programs.

4.1 Read generation

Read generation or simulation is the process of creating reads through the use of a "truth" genome and an error model. The goal is to generate reads which could have been plausible if the cell nucleus had contained the input truth genome. Also, as described in section 4.6, simulation provide knowledge of the input genome, which is used to evaluate the correctness of the results.

There are two approaches to read generation. One is to generate reads directly from the reference sequence, using a suitable error model, which is done in inGAP. The second approach is to split the process into multiple steps. Firstly, variants to be applied to a reference genome are chosen. Thereafter reads can be drawn from the modified reference, without mutations, but with sequencing errors according to an error model. This is done with GATK and GemSIM.

4.1.1 GATK FastaAlternateReference

The *walker* or program *FastaAlternateReference* in the GATK package is a tool for generating an alternative reference sequence, given an original sequence and a variant VCF file. It works for SNPs and indels, but not more complex substitutions. Also, if several variants exist at the same position, one of them is chosen at random. In the case of overlapping indels, only the first is chosen.

By applying a VCF file to the reference sequence, we have a "golden truth" list of variants, which can be compared to the variants found after simulation, mapping, re-alignment and variant calling.

Optionally, two different and independent VCF files may be applied to two copies of the reference genome, each generating one haplotype. For diploid organisms like humans, this is more realistic than simulating only one haplotype. Indeed, as shown in section 2.4.4, only about half of the variants are homozygous.

4.1.2 inGAP

[35] The *Integrated Next-gen Genome Analysis Platform (inGAP)* is mainly a java program for variant detection using a Bayesian algorithm. But in this section, it is more interesting to look at the additional tools it provides.

First, it can simulate reads, with or without quality scores, and with a given distribution on read length. Furthermore, it can incorporate random mutations to FASTA files, either SNPs, indels or both, and the exact frequency of the different mutations can be specified. However, the positions of the variants are random, and affects also unknown sections of the reference, characterized by long sequences of the character 'N'. SNPs are not added to these parts, but indels are.

The random mutations added are also not related to known mutations in dbSNP, and should not be used if this plays a role in downstream analyses.

Also, program source is not easily available, neither is proper documentation of the algorithms of the tools. There is no way to know if the parameters used by the program are based on empirical data from real sequencing platforms, and the read simulation functionality is not used in this thesis.

4.1.3 GemSIM

[29] The *General Error-Model based SIMulator (GemSIM)* is an open source python program dedicated to simulating reads, in particular from Illumina, both single and paired end, and from Roche/454.

The program takes a reference genome, an error model and optionally a haplotype file as input, and generates a simulated data set of random reads as could be produced by a sequencing run.

An error model for Illumina and Roche/454 is already included, and it is possible to use a supplied tool to generate an error model based on empirical data. Both single and paired-end reads are supported. Such error models predict the frequency of different read errors at a given position in a read, in a given context (3 preceding and 1 following base), of a certain type (SNP, insertion or deletion), and also the quality scores for true and mismatch bases.

The haplotype file is a way of simulating reads from a set of different sequences with known variations from the reference genome. Each sequence will be used with a given probability. This feature is particularly useful for *metagenomics*, but can also be used to simulate changes present in only one copy of the chromosome. GemSIM includes a tool to generate these haplotype files, though it only supports SNPs, not indels and structural variations. In this thesis only two different reference genomes from FastaAlternateReference have been used to simulate two haplotypes.

GemReads.py in the GemSIM package produces standard FASTQ read files which can then be processed further by mapping software.

4.2 Mapping

The mapping phase consists of aligning reads to a reference genome to create aligned reads. In the event of read errors or variants, there may be no exact matches, whereas repeated sections may yield multiple matches or *seeds*. Which program to use depends on read characteristics, like read length and probability of different kinds of errors. *BLAST* uses hash tables, and was one of the first effective aligners. It has later inspired other programs like *MAQ*, *SeqMap* and many others. More recently, programs based on suffix tries, like *BWA* and *bowtie*, have proved effective. A survey of algorithms and programs implementing these was written by Li and Homer [26], the main ideas are repeated here.

4.2.1 Hashmap

One strategy is to define patterns of positions, for instance k consecutive positions, but *spaced seeds* with holes are also possible. The reference genome is preprocessed. Then the pattern is applied to each read, and the haplotype from the pattern positions is compared with the reference.

Variants, in particular indels, may cause mismatches, therefore several positions on the read can be compared.

4.2.2 Tries and FM index

A suffix trie makes the inexact search more efficient, as all strings with the same prefix collapse to the same node in the suffix trie.

By using the Burrows-Wheeler Transform to create an FM index [3], it is possible to reduce the space requirement significantly, while maintaining a running time which is linear in the length of the query.

4.2.3 BWA

BWA, or *Burrows-wheeler Aligner*, uses two different algorithms, *ALN* designed for short and accurate reads [23] or *BWA-SW* for longer reads, more than a few hundred base pairs, with more errors [24].

Both algorithms are based on the Burrow-Wheeler Transform and suffix arrays, which has the advantages of trie matching, but with much lower memory requirements. This allows fast matching against a large reference genome, even in case of mismatches and gaps.

The *ALN* algorithm uses backtracking in case of inexact matching, which may be slow for reads with many errors. *BWA-SW* builds *FM-indices* for both the query and the reference sequence. It then applies dynamic programming to match the query to the reference. Unlike *BWT-SW* however, BWA-SW uses heuristics to prune bad matches. This greatly improves speed, but the true alignment is not guaranteed to be found.

BWA outputs the mapped reads in the SAM format by default, which can be converted to BAM files by SAMtools.

For Illumina reads, this is also the mapper recommended by GATK in their guide under the *best practices* section.

4.3 Realignment

As mapping algorithms consider each read independently [16], important information is omitted in the analysis. Data from variant databases can be used to generate *consensuses* or candidate haplotypes. But improvements may also be made by collecting and using data from multiple reads, or simply by doing a more thorough analysis than the mapper. This is feasible, because the number of suspicious mappings is small.

Realignment is usually done around a local realignment *window*, which allows more elaborate algorithms without huge running times. This is sensible, as many errors are local, for instance indels that are exchanged with SNPs.

There are two important algorithms in realignment:

4.3.1 Smith-Waterman

The Smith-Waterman algorithm [39, 36] is an algorithm for optimal alignment of two sequences, given a penalty for mismatches and gaps of various lengths. An optimal result is guaranteed, but the dynamic programming algorithm requires $O(N^3)$ time in its simplest form, and $O(N^2)$ time given affinity in gap cost, and is thus only done over a small window of the genome. Variant databases may be used to alter the cost function, but overlapping reads are not used.

4.3.2 Bayesian approach

The bayesian approach is named after Bayes rule, which states the relationship between conditional probabilities.

$$P(G|R) = \frac{P(R|G)P(G)}{P(R)}$$

To find the probability of a certain genotype G given data R, one has to estimate the other probabilities through knowledge about known variations and sequencing error characteristics. The Bayesian approach is in particular used for SNP discovery [8], and will typically use both variant databases and information from overlapping reads.

4.3.3 GATK IndelRealigner

The realignment algorithm in GATK has two steps. Theh first is to find mappings of low quality by running GATK *RealignerTargetCreator*. This step locates suspicious intervals which may be in need of realignment. Such intervals are so short that they do not impose a long running time.

The second step is where different consensuses are gathered in the GATK *IndelRealigner*, and the best one is chosen. The consensuses consist of reference genome with indels applied, and the best match is the one with the lowest *Hamming distance*. Both variant databases, indels in mapped reads and smith-waterman alignment can be used as a source.

4.3.4 Dindel realignment

Dindel uses four steps, where the first three are related to realignment. In the first step, called *getCIGARindels*, candidate indels are extracted from the provided BAM file. In case of paired-end reads, the insert size distribution is also calculated.

MakeWindows is the second step, where the indels from step one are grouped into realignment windows of approximately 120 bp.

The actual realignment algorithm in step three is the computationally most intensive step. Dindel uses the candidate indels from step one, together with SNPs present in the BAM files, to create candidate haplotypes. Finally the reads are realigned to these candidate haplotypes.

Dindel requires the use of an existing mapper to correctly map the reads to the correct region, the *realignment window* of 120 bp. Alternatively, dindel can use mate-pair information for all unmapped reads.

Once mapping has been done, dindel will generate different candidate haplotypes or sets of short DNA segments. With N variations, we get 2^N haplotypes, by using any subset of the variations.

The most probable candidate haplotypes (dindel uses 8) are tested against the read using a bayesian approach, where we calculate the maximum likelihood of a read R_i given a candidate haplotype H_j. The reference sequence is always among the candidates.

4.3.5 SRMA realignment

Short Read Micro-reAligner *(SRMA)* [16, 15] is a program used for better prediction of true variants through realignment. The main advantage, as stated by the article, is to reduce the false positive rate. As the name implies, only a very small window is considered for each read. The default is 20 bp in each direction.

All the inputs are put into a so-called variant graph. It contains all variants in all reads, both SNPs, insertions and deletions. The variant graph is a *directed acyclic graph* (DAG), where the nodes represent base pairs at specific positions relative to the reference genome, and edges represent neighboring base pairs.

SNPs can be thought of as parallel paths in the variant graph. Deletions can be thought of as edges that jump past a series of base pairs. Insertions on the other hand are like a path of extra nodes inserted between two base pairs.

The edges in the graph are weighted according to the probability of that edge being correct. There are several statistics involved in this, among others the number of reads and their read and alignment quality score. Reads are realigned using an algorithm similar to Dijkstra.

As an optimization, weak edges with low number *and* low proportion of reads can be pruned to save time, but should be done with care not to decrease sensitivity too much.

4.4 Variant calling

Variant calling or variant discovery is strongly linked with realignment and mapping. This is the stage where the information from previous steps is collected and evaluated in order to emit raw variants and qualities in a VCF file. It can involve simple interpretation of the output files from previous steps, but also complex and time-consuming generation and evaluation of variants.

4.4.1 GATK variant calling

There are two variant calling algorithms in GATK, the standard protocol *UnifiedGenotyper* and the experimental protocol *HaplotypeCaller*. In the GATK *best practices* guide, HaplotypeCaller is believed to be the best caller. However, it is marked as experimental, and not very well tested. UnifiedGenotyper is recommended as a fall back protocol.

UnifiedGenotyper uses a bayesian likelihood model to estimate the genotype and its probability in one or more samples. A known caveat is an aggressive calling strategy, resulting in a high false positive rate. The called variants may thus be in need of recalibration.

HaplotypeCaller combines a local de-novo assembler and an affine gap penalty pair *hidden Markov model*. It has the advantage of using the same likelihood model for SNPs, MNPs and indels. But the description of the algorithm is meagre, as well as the parameter recommendations, which awaits further experiments by GATK authors.

Both GATK variant callers use standard files as input which may be generated by other programs than from the GATK package. Thus it is easy to replace one stage in the pipeline.

4.4.2 Dindel variant calling

Dindel uses a very simple variant calling algorithm which just interprets the output from the realignment step, and produces indel calls and qualities in the VCF format. The variant calling algorithm is special in that it is tightly connected with dindel realignment. Furthermore, it is only designed to find indels, ignoring most SNPs. As most work is done in the realignment step, dindel variant calling is very fast.

Compared to other indel callers in *SAMtools* and *VarScan*, dindel had better results with respect to sensitivity and false positives[1]. In particular, dindel is claimed to be good at reducing the number of false positives.

To reduce computing times, the number of candidate haplotypes were restricted to 8. One can discuss if this kind of heuristic approach is a good one, but it is necessary in dindel.

Computing times is also one area where dindel performed poorly in the article, with quadratic time usage with respect to read depth.

It performed significantly worse on the real data sets compared to the simulated ones. This is thought to be caused partly by errors in the input data or in the capillary sequence data used as reference.

4.4.3 BCFtools variant calling

BCFtools is coupled with samtools for variant calling using a statistical framework. The samtools mpileup command collects information from the input BAM file and computes the genotype likelihood given some assumptions [22]. BCFtools then calls variants and gathers various statistics. The two steps are separated by a special BCF file passed between the two programs.

BCFtools does not properly handle multi-allelic variants. It only takes the strongest non-reference allele.

4.4.4 VarScan 2 variant calling

VarScan 2 also depends on samtools mpileup, but with a different intermediate file format, namely a pileup file.

Instead of the bayesian method employed by many other variant callers, VarScan 2 uses a heuristic approach where various factors are taken into account, followed by a statistical test.

Each position in the pileup file is parsed, and VarScan decides whether all requirements for calling a variant are present. Many of these parameters can be changed at the command line. This is very important, especially if input is uncommon, for instance, a coverage of 8 is the default minimum threshold. There are similar limits for variant frequency, read quality, number of supporting reads and other tests.

4.5 Variant recalibration

After different programs have generated variant files, a key question is whether the called variants are correct. Some variant callers, like in GATK, tend to be quite aggressive, not to miss many true variants. Consequently many sequencing artifacts are reported as variants, resulting in a high number of false positives. This is undesired.

There are two training sources for recalibrating variants. One is the reads and variant files themselves, from which we can extract quality scores and metadata. The second source is variant database files. True variants will be overrepresented in the variant database, whereas false positives will have a rate approximately proportional to the size of the database only.

From these sources we can train the recalibrator to filter away false positives, while keeping as many of the true positives as possible. But ultimately it is a tradeoff between sensitivity and specificity, and the user must prioritize.

4.5.1 GATK variant recalibration

Recalibration in GATK is at least a two-step process. First *VariantRecalibrator* needs to be run once (for HaplotypeCaller) or twice (for UnifiedGenotyper, one for SNPs and one for indels). Afterwards the generated error model is used as input for *ApplyRecalibration* which does the actual filtering of variants. In the last step, the user is required to decide on a preferred sensitivity, which indirectly decides the specificity.

4.5.2 Quality threshold

A very simple recalibrator is one that discards all variants below a certain quality threshold. As all metadata are ignored, this approach is expected to be suboptimal. Nevertheless, it is very simple to implement, and it provides valuable information on the correctness of quality scores.

All variant callers used in this thesis provides quality scores, except VarScan 2, which instead uses a p-value in the info field.

4.6 Variant validation

If this was a biological sample, this is about as far as we get. There is no way to know if a given variant was actually present in the sample. We can always refine the results by drawing more samples, but in the end, all we have is an estimate.

With simulated reads we have a golden truth, namely the input to the simulator. The set of true positives is exactly the intersection between the called variants and the simulator input. Similarly false positives are the set of called variants, except those present in the simulator input. False negatives are the set of simulator input except those in the set of called variants.

However, the problem is harder than comparing lines in files. What we ultimately would like to know is the two haplotypes. Variant files generate only the genotype, which is slightly less informative (see section 1.1.6). Nevertheless, with access to the reads, information may be recreated. Another problem arises with the different representations of variants. The easiest example is MNPs which may be represented as a series of SNPs. But indels can also be represented in different ways, some require even change of more than one line to switch from one representation to the other. This is described in section 4.7.

We have more or less ignored the first problem, assuming that all matching heterozygous variants belong to the correct haplotype. But the second problem should be solved. Neither VCFtools, nor HTSlib will check the underlying nucleotide sequence in case of mismatches.

4.6.1 VCFtools

A useful program for comparing and manipulating VCF files is *VCFtools* [9]. For verifying correctness of VCF files, we can use *vcf-compare*, which takes two or more VCF files as input, and output the number of variants present in any intersection of the input files and their complements. This information can be used directly to create a Venn diagram. Various tools also exists for VCF file manipulation, including merging and intersection.

4.6.2 HTSlib

The VCFtools homepage recommends the C library HTSlib as an alternative for specific VCFtools commands. This applies in particular to merging and intersection of vcf files, and extracting statistics. The latter takes up to two variant files as input, and displays statistics on the intersection and the relative complements. The number of occurences in each of the sets are reported, as well as detailed breakdown into indels of all lengths and all the different substitutions in SNPs.

4.6.3 Different variant types

From the source code of HTSlib we can see the definition of the different variant types. SNPs are, as expected, single substitutions. For all other types, we first ignore the longest common prefix. If we are left with either the reference or alternative being empty, we have an indel. If both are non-empty, we have a complex substitution which is either a MNP if the length is equal, or "others" otherwise. In other words, a MNP is 2 or more consecutive SNPs, whereas others is a combination of a SNP or MNP and an indel. Others may also refer to misformatted variants.

If a variant file has multiple comma-separated alternatives for a single variant, this adds to the number of "multiallelic sites", this can for instance happen with heterozygous variants.

4.6.4 Comparison of vcf-compare and htslib

The advantage with htslib is speed and its very detailed statistics. vcf-compare, on the other hand, can take any number of files as input which makes it easy to compare the discovery rate of both homozygous and heterozygous variants.

It is, however, just a matter of taste, as variant files may be merged or intersected to create any set of interest for htslib. And variant files can be filtered on a specific variant type to create detailed statistics for vcf-compare.

4.7 Aligning variant files

As section 4.6 tells us, different representations are a problem that need to be solved, not only as a theoretical problem, but also something that arises in practice. An extract from dbSNP gives us these lines:

```
1 70484 rs149559312 CTCTT C
1 70490 rs140218451 C T
...
1 77201 rs147208506 ACA A
```

whereas an extract from a variant file from GATK yields:

```
1 70485 . TC T
1 70487 . TTTC T
...
1 77200 . AAC A
```

Chromosome number is changed from 20 to 1, and irrelevant information is omitted. Apart from that, the information is copied as it is.

If we look at positions 77200-77203, we see that in both cases, the reference genome AACA is changed to AA. But the representation is different; they start at different positions, 77201 and 77200. This case can be fixed by left-aligning the line from dbSNP.

The second case, the two upper lines in each variant file extract, is more complex. Here, the positions 70484-70490 with reference CTCTTTC are replaced with CTT. But all four variant lines are valid and left-aligned.

In both cases, the haplotypes are equal for dbSNP and GATK, but variant validation tools erroneously report mismatch. Not only does it result in less true positives, but also increases the number of both false positives and negatives.

The first case can be fixed by left-aligning all variants, while the second is more complex, and a possible solution is to try to convert from one form to the other.

The second case can be extended to a third case, where variants are equivalent, but not overlapping. This happens typically due to repetitions in the genome, and represents a challenge in variant aligning, because the window of interest may be quite large.

4.7.1 GATK LeftAlignVariants

This walker is a simple tool that takes a reference file and a variant file as input, and outputs the left-aligned version of the variant file. It does also fix other issues, for instance INFO fields not in sorted order. It does not, however, fix errors occurring due to complex cases.

4.7.2 All-subset comparison

In addition to the available tools, we also wrote a python script doing alignment of multiple variants, found in appendix 9.1.3. The goal is the same as above, to alter a trial variant file to have as much as possible in common with a truth variant file, without changing the underlying haplotype(s).

Whereas LeftAlignVariants only looks at a single variant at a time, there are many occasions where multiple variants need to be changed in one operation.

Common algorithm

The strategy used is to loop through each variant in the trial file, ignoring all variants which are already matched in the truth file and cannot be improved.

For each such mismatching variant, we search trough nearby variants and see if they can be added to create two equivalent variant sets. Both the trial set containing the mismatching variant and the truth set must be generated, and both must create identical haplotypes when applied to the reference genome.

If such a set is found, the truth set can be added to the list of true positives, whereas the trial set is removed from the false positives. If there are matches in the trial set against the truth file, these true positives are not removed, as they could belong to the other haplotype in a diploid genome.

The main problem is to find if there is a truth set and trial set which is different, but equivalent.

Dynamic set extension

By dynamic set extension we order the variants by position. For each new variant, we may either add it to the relevant set, or we may ignore it. In either case, we extend the variant set haplotype by at least one base pair. If the truth and trial haplotype mismatch, we may stop immediately. If they match, we have found what we are looking for, and may also stop. However, if they match up to a certain point, we may need to add more variants to either set and repeat the process.

The advantage is that most sets are discarded fast, reducing the number of sets that need to be considered.

The disadvantage is that one needs to find the first variant, that with the lowest position in the reference. This is because an indel will shift all following nucleotides, which severely complicates the algorithm. Furthermore, the variant can be both in the

truth set and the trial set, thus it is impossible to only iterate over the variants in the trial set.

Brute force set testing

A key observation is that variants are rare. Between 0.1 and 0.15% of the positions contain a variant. Thus in a window of 2000 nucleotides, we expect 2-3 variants. Of course this applies to both the truth and trial file, which also tend to be on similar places. Furthermore, this is only the average number, and the maximum is much higher.

The total number of subsets that can be drawn from X variants are 2^X, or exponential growth. It is, however, manageable for small X. Furthermore, many of these subsets can be discarded immediately, because the length of the indels does not sum up to the same value. One haplotype has a shift relative to the other, and they cannot be equivalent.

The rest can be checked by brute force. Both haplotypes are generated, and any difference results in a mismatch. Ideally the mismatch is discovered immediately, but in the worst case, one can generate the whole window first and then compare. That would still be feasible due to the low variant frequency and limited window size.

In the program, a window size, X, is chosen as a parameter. It will then generate all subsets of variants in the trial file starting between position A and $A + X$, where A is the position of the mismatching variant from the loop.

In the truth file, we use all subsets of variants starting between position $A - X/2$ and $A + X + X/2$. This way we avoid the problem described in the previous part about finding the first variant, but the region to search is larger.

Finally there is a security valve that will cut the interval size in half if the total number of variants exceed a specified limit, for instance 25, as 2^{25} is still feasible.

Future improvements could include more efficient cutoffs of non-equivalent sets or memoization of incompatible variants. Another possibility is to remove variants already matching in the truth file, but this should be done with care, especially with diploid genomes.

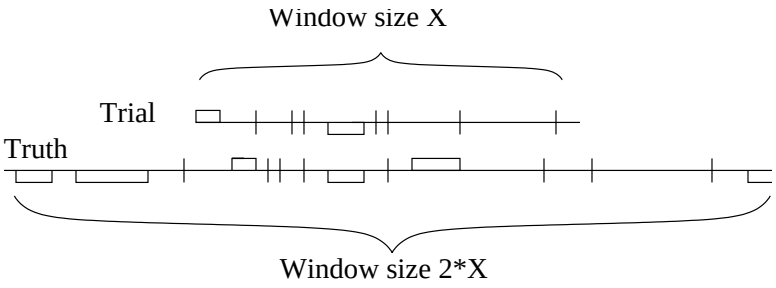

Figure 2: Illustration of windows used by brute force set testing. Note that the first (left) variant in the trial file must be present in the set, and also not exist in the truth file. Indels represented as boxes and SNPs as lines.

Diploid genome

With diploid sequencing, there may be variants that are only present in approximately half of the reads. Such heterozygous variants from different haplotypes should not be combined in a set.

Variants from variant callers may be marked with frequency, which indicates if it is heterozygous or homozygous, but not with haplotype. However, the truth file generated by merging two variant files contains haplotype information, and it is easy to force heterozygous variants from different haplotypes not to be used in the same variant set.

4.7.3 Correct VCF file by variant calling

An alternative way of correcting a VCF file is to use the variant callers to create truth files. Instead of using a simulator to generate raw, plausible reads, we may make a special simulator that creates high quality mapped reads.

The hypothesis is that the representation will be equal when both the truth and trial variant file are generated by the same variant caller. In addition, if input to variant calling is error-free with sufficient coverage and quality score everywhere, all variants should be present in the truth file.

Nevertheless, this approach was not tested, and is left to later studies.

4.8 Coverage

The average coverage is easily calculated by dividing the number of simulated bases (which is the number of reads times the read length) with the length of the reference. But it is also very important to know something about the coverage at certain locations at the genome. With such information, it is possible to study the importance of coverage for variant calling and thus find a good trade-off between correctness and resource cost.

It is possible, but slightly difficult, to alter the source code to do this in the simulator. All read positions can be written to a log file, before the position is converted from haplotype coordinates to reference coordinates used by the variant files. The conversion must use the set of indels and complex variants for this.

An easier solution, which is used in this thesis and described below, is to use mapping output.

4.8.1 samtools depth

The samtools *depth* tool counts the coverage at every position in the aligned reads. With this approach we get the assumed coverage by the mapper or aligner. The coordinates origin from the reference file and can thus, easily be mixed with data from variants files.

Note that deletions cause the depth do drop, even if the read span both sides of the deletion.

Positions with zero depth are omitted in the output file, unless this is due to deletions. For areas with unexpected low coverage, it can be helpful to look at the coverage in a small window on both sides, and if both are high, it is likely that the drop is caused by deletions, not gaps between reads.

4.8.2 GATK CoverageBySample

GATK also has a program to calculate coverage, but this was very slow and did not output position in reference, hence it was not used.

4.8.3 Variant coverage

To the author's knowledge, there exist no tools for filtering variant files according to coverage. Thus a simple python script was created, using samtools depth as input.

Normally the depth data are used directly, but if the coverage is higher in a window of 10 bp to both sides, the highest depth from the lowest window is used instead. With this method, the reported coverage is never reduced.

This approximation is used to prevent mistakes from deleted areas. However, deletions may be much longer than 10 bp, so the solution is not ideal. But the position reported in the variant file refers to the first bp in the ALT string, the one that is not removed, thus in theory this should not be a big issue.

The variants in the input file are distributed to the output files according to the altered depth calculation. With intervals, the number of output files may be reduced, which is good for summarizing data.

5 Methods

In this section we present the experiments, one in each subsection. There are three whole-pipeline experiments, each corresponding to a different data set. Then there is one experiment about variant representation and correction, and one experiment about recalibration.

Each subsection describes the programs used, with possible variations and parameters. The data from these experiments are presented in section 6.

Many experiments were conducted based on the results from earlier experiments, and thus the complexity of the experiments tended to increase over time.

Finally, there is a subsection describing the computing environment used.

5.1 Validation of variation calling

This first experiment was conducted to verify correctness and accuracy of the different algorithms. Important hypotheses were to see if the different variant callers actually work and if they find a significant share of the variants. Furthermore we investigated what kinds of variants were harder to find, and which reported variants were most likely to be wrong. In other words, we studied the frequency of false negatives and positives.

In the process, we also studied read generation, mapping, realignment and variant validation. For this first experiment, only a haploid chromosome with low-coverage reads was used.

Both in this and in the following experiments, we limited the data to chromosome 20, as it is relatively small and contains relatively few unknown base nucleotides in the reference genome.

5.1.1 Experiment overview

There were multiple steps in this experiment, it can be roughly categorized in four tasks. First was to generate a genotype that was both a source of reads and a golden truth. Second was to generate the reads by simulation. Third was to run the entire pipeline, from mapping to realignment and variant calling. Fourth was to compare the results with the golden truth.

To generate a genotype, we used variant database files to extract a plausible set of variants, according to the statistics in section 2.4.4. These were applied to a reference genome to generate two haplotypes, which together constitute one genotype.

In this experiment, both haplotypes were equal, corresponding to a haploid genome, or equivalent, a genotype with only homozygous variants. This is not a realistic scenario for real-life samples, but should make variant calling easier.

From the genotype, random reads were drawn, and quality scores and sequencing errors were added according to an empirical error model. These reads were processed by several algorithms in a pipeline, from mapping through realignment and recalibration to variant calling and filtration.

Finally, the final variant set was compared to the truth variants used to generate the haplotypes. The variant files may also be split into SNPs and indels, to study the quality of different kinds of variant calling.

5.1.2 Coverage

A total of 3 million single-ended illumina reads of length 101 were simulated. With a total number of 63025520 bases, of which 59505520 are regular, this gives an average

coverage of $\lambda = 3.03 * 10^8/63025520 = 4.81$. However, as the starting point for each read is determined in the simulator by a random number generator, the coverage will be lower in large parts of the genome. Some areas may even be completely uncovered, rendering variant calling impossible.

A hypothesis is that the coverage of a given position follows a poisson distribution, because the number of reads is high and the reference is long. There is a weakness with this hypothesis, because each read covers 101 continuous positions, not just a single position. Therefore the coverage of a given position is strongly correlated with the neighboring positions.

Because reads that are not contained in the chromosome are not included, the coverage is lower at the beginning and end. But these positions are not sequenced and contains no variants anyway.

5.1.3 Generation of variant files

First, chromosome 20 is extracted to have a small subset of variants to work with. This can be done with a trivial script that removes all lines from the VCF file with the wrong chromosome number. We also rename chromosome 20 to chromosome 1 in all lines for technical reasons.

This way we generate a 163 MB file `dbsnp.20.vcf` from the 7 GB `dbsnp_135.b37.vcf`.

To generate relevant statistics we can run another script:

```
$ generateVCFStatistics.py dbsnp.20.vcf dbsnp.20.stats
```

Note that the script which generates statistics will truncate all but one random variant from each line with multiple variants. However, this sampling error is negated by similar behaviour in the algorithm that picks random variants, which is done by

```
$ generateTestVCF.py dbsnp.20.vcf sample.20 dbsnp.20.stats
```

This selects two variant sets, A and B which are similar to the true frequency of variants, both SNPs, MNPs and indels. To simulate both homozygous and heterozygous variants, A and B are overlapping significantly, also according to statistics in section 2.4.4. Furthermore, a fraction of the variant in A and B is removed from the database, to simulate novel variants. In these experiments, 25% of the variants were novel. The remaining variants in the database are put in the file with DB in the name.

It should be noted that in the first experiment, only a simplified version of variant set A was used, with only SNPs and simple indels.

Three files are generated, `dbsnp.20.{A,B and DB}.vcf` After compressing the files with `bgzip file.vcf` and indexing them with `tabix -p vcf file.vcf.gz`, we can check the overlaps with vcf-compare.

```
$ vcf-compare sample.20.A.vcf.gz
sample.20.B.vcf.gz sample.20.DB.vcf.gz
```

Instead of using vcf-compare, one can use htslib to obtain statistics from one or two variant files. This tool automatically output statistics on SNPs, MNPs, indels and complex variants.

```
$ htslib vcfcheck sample.20.A.vcf.gz sample.20.B.vcf.gz
```

5.1.4 Generation of reads

First step in read generation is to extract chromosome 20 and create alternative FASTA files corresponding to the haplotype:

```
$ java -Xmx4g -jar GenomeAnalysisTK.jar -R human_g1k_v37.fasta
  -T FastaReferenceMaker -o human_g1k_v37_chr20.fasta -L 20

$ java -Xmx4g -jar GenomeAnalysisTK.jar -R human_g1k_v37_chr20.fasta
  -T FastaAlternateReferenceMaker
  -o haplo.A.fasta --variant sample.20.A.vcf
```

Next, we draw $3 * 10^6$ random single-ended reads from the haplotype.

```
$ GemReads.py -r haplo.A.fasta -n 3000000 -l d
  -m gemsimpath/models/ill100v5_s.gzip -q 64 -o simreads
```

The read generator does not include any information of the position in the input file, thus it cannot spoil the results by giving certain algorithms additional information.

5.1.5 Mapping stage

For the mapping stage, the algorithms *aln* and *samse/sampe* in BWA are used. Note that samse only works on single-ended reads as the name implies, whereas sampe is used for paired-ended reads.

```
$ bwa aln human_g1k_v37_chr20.fasta
  simreads_single.fastq > tmp/simreads.sai

$ bwa samse human_g1k_v37_chr20.fasta tmp/simreads.sai
  simreads_single_fastq > tmp/simreads.sam
```

The mappings are then converted to binary, sorted, indexed and assigned read groups that are necessary for GATK in the next step:

```
samtools view -bS tmp/simreads.sam > tmp/simreads.bam

$ java -jar picard-tools-1.77/AddOrReplaceReadGroups.jar
  I= tmp/simreads.bam O= simreads.bam SORT_ORDER=coordinate
  RGID=1 RGLB=lib_1 RGPL=illumina
  RGSM=1 RGPU=simulated CREATE_INDEX=True

samtools index simreads.bam
```

5.1.6 Realignment

As described in section 4.3.3, the mapped reads can be sent through an optional realignment phase consisting of two steps. The default model was used, namely to use both variant database and mapped reads as a consensus source. It is possible to extend IndelRealigner by using smith-waterman alignment with the argument `-model USE_SW`, or to restrict the model to only use indels from a list of known database indels with `-model KNOWNS_ONLY`. The model parameter is added to the walker *IndelRealigner*.

```
$ java -Xmx4g -jar GenomeAnalysisTK.jar -R human_g1k_v37_chr20.fasta
  -T RealignerTargetCreator -I simreads.bam -o tmp/simreads.intervals
  -known sample.20.DB.vcf
```

```
$ java -Xmx4g -jar GenomeAnalysisTK.jar -R human_g1k_v37_chr20.fasta
  -T IndelRealigner -I simreads.bam -targetIntervals
  tmp/simreads.intervals -known sample20.DB.vcf
  -o simreads.realigned.bam
```

The other realigner described, srma (see section 4.3.5) only requires one command:

```
$ java -jar srma-0.1.15.jar I=simreads.bam
  O=simreads.srma.bam R=human_g1k_v37_chr20.fasta
```

It is possible to use a `RANGE=chr:start-end` parameter, but the reads must then be merged afterwards. This is suggested on the wiki page of SRMA. There is also the option to use the `GRAPH_PRUNING=Boolean` parameter.

5.1.7 GATK variant calling

The next step in the pipeline is to find the actual variants. There are two algorithms in use today by GATK, the *UnifiedGenotyper* and *HaplotypeCaller*. These algorithms can be applied to the reads from the mapping stage, or those that also have been through the realignment stage. Output files are marked with `.realigned.`, `.unrealigned.` or `.srma.` to specify which input BAM file was used.

Both algorithms output indels as well as SNPs. This behaviour can be changed in UnifiedGenotyper by modifying or removing the *-glm BOTH* parameter, but the exact same result can also be achieved by post-filtering the vcf file.

For UnifiedGenotyper, the arguments were:

```
$ java -Xmx4g -jar GenomeAnalysisTK.jar -R human_g1k_v37_chr20.fasta
  -T UnifiedGenotyper -glm BOTH -I simreads.realigned.bam
  -stand_call_conf 30.0 -stand_emit_conf 10.0
  -o simreads.unified.realigned.vcf -nt 8
```

Whereas HaplotypeCaller used the following arguments

```
$ java -Xmx4g -jar GenomeAnalysisTK.jar -R human_g1k_v37_chr20.fasta
  -T HaplotypeCaller -I simreads.realigned.bam
  -stand_call_conf 30.0 -stand_emit_conf 10.0
   -o simreads.haplo.realigned.vcf
```

This gave the following 4 variant files:

```
$ simreads.(haplo|unified).(realigned|unrealigned).vcf|
```

The two algorithms did not only produce different variant sets, they also had very different running times. UnifiedGenotyper, when run with 8 simultaneous cores needed less than 5 minutes. HaplotypeCaller does not yet allow multithreading and ran for 16-20 hours.

5.1.8 Dindel variant calling

An alternative to the variant calling in the GATK pipeline is the program *dindel*. As the name implies, it is only used for finding indels, and does so in a number of steps as described in section 4.3.4 and 4.4.2.

```
$ dindel-1.01-linux-64bit --analysis getCIGARindels --bamFile
simreads.realigned.bam --outputFile simreads.realigned.dindel_output
--ref human_g1k_v37_chr20.fasta
```

```
$ makeWindows.py --inputVarFile
simreads.realigned.dindel_output.variants.txt --windowFilePrefix
tmp/simreads.realigned.dindel_realign_windows
--numWindowsPerFile 1000
```

As makeWindows.py creates multiple files, the next command must be run once for each of the generated files, as it is shown for file number 12.

```
$ dindel-1.01-linux-64bit --analysis indels --doDiploid --bamFile
simreads.realigned.bam --ref human_g1k_v37_chr20.fasta
--varFile tmp/simreads.realigned.dindel_realign_windows.12.txt
--libFile simreads.realigned.dindel_output.libraries.txt
--outputFile tmp/simreads.realigned.dindel_realign_windows.12.txt
```

Finally the *.glf.txt* files that are generated are listed up in a text file, for instance by using the standard unix ls command

`/bin/ls tmp/*glf.txt > tmp/realigned.dindel_stage2.list`

This file is then used in the final step:

```
$ mergeOutputDiploid.py --inputFiles
tmp/simreads.realigned.dindel_stage2.list --outputFile
simreads.realigned.dindel.vcf --ref human_g1k_v37_chr20.fasta
```

5.1.9 VarScan 2 variant calling

VarScan is described in section 4.4.4, and works in two steps. The first is to generate a pileup file from samtools mpileup:

```
$ samtools mpileup -f human_g1k_v37_chr20.fasta simreads.realigned.bam
> tmp/simreads.realigned.mpileup
```

From the pileup file, varScan will generate a consensus, from which variants will be drawn by using `--variants`. With `--output-vcf 1`, these are converted to VCF format.

```
$ java -jar VarScan.v2.3.3.jar mpileup2cns
tmp/simreads.realigned.mpileup  --output-vcf 1 --variants
> simreads.realigned.varscan.vcf
```

5.1.10 Samtools/bcftools variant calling

For samtools/bcftools we use the same mpileup algorithm as in VarScan 2, but with a `-g` parameter to output in BCF instead of pileup format. Normally `-u` is used instead of `-g` when piping output directly.

```
$ samtools mpileup -gf human_g1k_v37_chr20.fasta simreads.realigned.bam
> tmp/simreads.realigned.mpileup.bcf
```

Bcftools does the actual variant calling.

```
$ bcftools view -vcg tmp/simreads.realigned.mpileup.bcf
> simreads.realigned.bcftools.vcf
```

5.2 Variant representation

In this experiment we investigated the impact of different variant representations, and how this problem could be fixed by variant correction. Both the impact on SNPs and indels was investigated.

5.2.1 Experiment overview

In this experiment, only variant files from the previous experiment (section 5.1) were studied.

The data from the first experiment indicated that there might be issues with different representation of indels (see section 6.1.4). And deeper study into the variant files revealed examples like the complex case in section 4.7. This is problematic because it gives a wrong image of the accuracy of the variant callers, and also penalizes programs which have a non-standard representation.

First we tried to change the variant representation. We then compared the number of false positives and negatives with the uncorrected data, and tried to find interesting differences. Finally we verified the variant files, that they were in fact equivalent and represented the same haplotype.

5.2.2 Change variant representation

For this part we use the self-made tool described in section 4.7.2. As a trade-off between execution speed and completeness, a window size of 500 was used. This means a window of 500 for the trial files and 1000 for the truth file. The program tests all subsets of variants in these windows.

The usage is

```
$ python correctvcf2.py truth.vcf trial.vcf
  reference.fasta [corrected.vcf] [window size]
```

Unless otherwise noted, the left-aligned (see section 5.1) version of the truth file, `sample.left.20.A.vcf`, was used. The trial files are output from the different variant callers.

GATK FastaAlternateReferenceMaker can be used as in section 5.1.4 to verify that the variant files before and after correction are equivalent. This works only on a haploid genome, as we have in this case, or if we had known which haplotype the variants belong to.

The last step, to study the remaining mismatches, was done by manual inspection of the variant files, after filtering out true positives and variants of a specific type using vcf-isec in VCFtools:

```
$ vcf-isec -c variant_in_this.vcf.gz except_these.vcf.gz > out.vcf
```

5.3 Better coverage

In this experiment, we used a second data set with four times the number of reads, though still haploid and single-ended. This was to verify if the findings were still true with higher coverage, or if the preferred algorithm changes with the coverage.

Another question was whether there is a correlation between coverage, measured coverage and variant calling accuracy, and how strong this correlation is.

5.3.1 Experiment overview

The method of data generation is, if nothing else is stated, exactly as in the first experiment in section 5.1. A major difference is the number of reads, which were increased to 12 million reads of length 101, or $1.212 * 10^9$ bp. This gives an average coverage of 19.23

The variant files were identical, in particular there were no MNPs or complex variants in this experiment either.

Also some of the less promising algorithms have been omitted. This includes the SRMA realigner, mostly due to the run-time issues.

5.3.2 Coverage distribution

If we assume poisson distribution, with average 19.23, the areas with low coverage are now very few, in contrast to the data from section 5.1.2. In fact there are less positions with expected coverage of 9 or less, than there were expected uncovered positions in the previous experiment.

In addition to the theoretical coverage, we may instead look at the depth of the mapped reads as in section 4.8.1.

```
$ samtools depth simreads.realigned.bam > simreads.realigned.depth
```

To count the occurences of all different read depths across the whole file, one may use the standard unix tools `sort` and `uniq`, where we specify which column to group by, and to use numeric sort. The number of occurences are in the first column of the output.

```
$ cat simreads.realigned.depth | sort -n -k 3 | uniq -f 2 -c
```

5.3.3 Correlation on coverage and variant calling

A natural next step is to split the truth variant file according to the read depth at the location of the variant. For this we use the data from samtools depth, and sort the entries of the variant file according to the depth. Afterwards we can find the number of false negatives and true positives in each group by comparing with the truth file.

As no tools were found to do this job, a simple script was created as described in section 4.8.3.

```
$ python variantdepth.py simreads.realigned.depth
  sample.simple.left.20.A.vcf 5
```

Note that the depth file is calculated from the BAM file from the mapper or realigner, whereas the variant file is from the variant caller.

The generated files were zipped and indexed, and compared with the truth file by htslib vcfcheck.

5.4 Diploid genome

In this experiment, we changed the data set to a diploid genome with complex variants as well as simple indels and SNPs. Pair-ended reads were used instead of single-ended, and average coverage was also higher than before, for both haplotypes.

We also studied the difference between heterozygous and homozygous variants, how this affected the sensitivity.

Only the best approaches from the previous experiments were tested; thus only GATK and Dindel variant callers were used, and variant correction was done on all variant files. In addition, we merged the variant files from different variant callers to find if there were any improvements in sensitivity.

5.4.1 Experiment overview

First variant files as in section 5.1.3, with frequencies as in tables 2 on page 54 and 3 on page 54 were used to generate haplotypes. From these, we drew paired-end reads which then was mapped with bwa and realigned with GATK IndelRealigner, before different variant callers were used. The variant files were thoroughly analysed to find different characteristics of the results.

5.4.2 Generation of reads

Where nothing else is noted, the generation of reads was done similarly as in section 5.1.4.

A second haplotype, haplo.B.fasta was generated from the second variant file.

```
$ java -Xmx4g -jar GenomeAnalysisTK.jar -R human_g1k_v37_chr20.fasta
  -T FastaAlternateReferenceMaker
  -o haplo.B.fasta --variant sample.20.B.vcf
```

Next, we drew 10^7 random paired-ended reads from each haplotype, each with length 101, giving $4.04 * 10^9$ base pairs.

```
$ GemReads.py -r haplo.A.fasta -n 10000000 -l d -p -u d
  -m gemsimpath/models/ill100v5_p.gzip -q 64 -o simreads.A
```

And similarly for the second haplotype.

5.4.3 Mapping stage

Mapping differs from section 5.1.5 in that we have paired-end diploid reads.

The first step is the same, but repeated four times; for both ends and both haplotypes:

```
$  bwa aln human_g1k_v37_chr20.fasta
   simreads.A_fir.fastq > tmp/simreads.A_fir.sai
$  bwa aln human_g1k_v37_chr20.fasta
   simreads.A_sec.fastq > tmp/simreads.A_sec.sai
$  bwa aln human_g1k_v37_chr20.fasta
   simreads.B_fir.fastq > tmp/simreads.B_fir.sai
$  bwa aln human_g1k_v37_chr20.fasta
   simreads.B_sec.fastq > tmp/simreads.B_sec.sai
```

These were then mapped, for both haplotypes:

```
$  bwa sampe human_g1k_v37_chr20.fasta
   tmp/simreads.A_fir.sai tmp/simreads.A_sec.sai
   simreads.A_fir.fastq simreads.A_sec.fastq
   > tmp/simreads.A.sam
$  bwa sampe human_g1k_v37_chr20.fasta
   tmp/simreads.B_fir.sai tmp/simreads.B_sec.sai
   simreads.B_fir.fastq simreads.B_sec.fastq
   > tmp/simreads.B.sam
```

These mapped reads were then converted to BAM format and merged, in order to mix reads from both haplotypes in the same file. Because BWA looks at one read pair at a time, this should give the exact same result as if reads from both haplotypes had been merged first and then mapped.

```
$  samptools merge tmp/simreads.bam
   tmp/simreads.A.bam tmp/simreads.B.bam
```

Finally read groups and index were added just like the previous experiment.

5.4.4 Realignment and variant calling

The standard GATK realignment was done as in section 5.1.6, with the default model, *USE_READS*, in the IndelRealigner. SRMA was not used, due to the run-time problems.

In addition, dindel realignment and variant calling followed the instructions in section 4.4.2 exactly.

Only the most promising variant callers were tested, those being dindel, GATK UnifiedGenotyper and HaplotypeCaller, with parameters as described in section 5.1.7. Input were in all cases realigned mapped reads.

5.4.5 Advanced variant analysis

To summarize information from multiple files, we may merge variant files to contain the union of the original files:

```
$ vcf-merge sample.20.A.vcf.gz sample.20.B.vcf.gz >sample.20.AB.vcf
```

It is also possible to perform standard merge using vcf-isec, by using the parameter `-n +1`, which specifies that a variant must be present in at least one input file.

```
$ vcf-isec -n +1 sample.20.A.vcf.gz sample.20.B.vcf.gz
  > sample.20.AB.vcf
```

vcf-isec can also find the intersection of variant files, alternatively find the intersection between a variant file and the complement of other files.

The merge file will contain all sites with variants in at least one of the input files, and also mark in the INFO field SF which source file(s) were used.

For this experiment it is particularly interesting to look at the union of output files from variant callers. Can the results be improved if we combine the results of different callers?

5.4.6 Coverage distribution and variants

As in section 5.3.2, we can calculate the coverage distribution of the reads, there is no difference in the command when run on a diploid genome. Later we can run the analysis in section 5.3.3, and, as the data are diploid, it is interesting to see if there is any difference between heterozygous and homozygous variants.

Instead of splitting the truth file according to coverage first, and then analyzing the overlap with the variant caller file, we can simplify the data collection. First we calculate the intersection between variants and truth file to find the set of true positives. Then this set can be split according to coverage. The number of true positive variants with a certain coverage is just the number of lines in the corresponding file, minus the header section. Other intersections can be used to find number of false negatives with a given coverage.

5.5 Variant recalibration

In this final experiment, we investigated the important trade-off between precision and sensitivity, described in section 4.5. Different variant callers were tested against each other, to see if the results would be different if the aggressiveness parameters were changed.

Both the medium-sized and large data set was investigated, on both SNPs and indels.

5.5.1 Experiment overview

We used the variant caller output from UnifiedGenotyper, HaplotypeCaller and Dindel, the latter only for indels.

Different thresholds on variant quality scores were set, and the effect on precision and sensitivity observed. These data were plotted together in the same graph, in order to compare variant caller performance.

5.5.2 Generation of sensitivity/precision graph

We sorted the variant file according to calling quality, and annotated the variants with "correct" if they were found in the truth file, and with "false" otherwise. The SNPs and indels were extracted to create separate statistics.

Then we could easily set the variant quality threshold to any value, with a corresponding filtering of variants. Thus we got sensitivity and precision for any aggressiveness level, or equivalently, for any level of sensitivity below the maximum.

This was achieved with a self-written script that can be found in appendix 9.1.5.

```
$ python sensitivitygraph.py truth.vcf corrected.trial.vcf trial
```

This line generated two files, `trial.snp_statistics` and `trial.indel_statistics`, which could then be plotted using any standard plotting program.

5.6 Environment setup

Unless otherwise specified, all timed programs were run on a single computer (Red Hat Enterprise Linux Client release 5.8 (Tikanga)) with 8 GB of memory and 8 CPUs (Intel(R) Core(TM) i7 CPU 870 @ 2.93GHz). Data files were stored on the *Abel* computer cluster at the University of Oslo and accessed via *sshfs*.

An alternative would be to run some or all programs on the computer cluster, in particular for programs that support multi-threading and take a long time to run. This will also yield more consistent results, than a computer simultaneously used for browsing and writing (by a single person). The disadvantage is the administration and overhead due to the queueing system on the cluster, where tasks are put in a queue, scheduled to run at a later time, and abruptly killed if exceeding the allotted time window.

All input files and programs are assumed to be in the same folder, alternatively included in the PATH. Where this is not true, commands may need to be changed accordingly.

File sizes are measured using binary prefixes (powers of 1024) and rounded up.

5.6.1 Timing errors

Due to inefficient data management, where files are stored on an external server, time usage may not be accurate. File intensive algorithms will be slower than usual, and moving to a supercomputer with efficient communication and much memory can drastically reduce running times.

Furthermore, by using a shared computer, the processes may be swapped out regularly, especially under heavy load.

Thus all timing information should be taken with a large grain of salt, and only used as a vague indication on the running time.

6 Results

In this section we will present the results and timings achieved by running the different programs from section 5. Possible implications of the results are also discussed.

6.1 Validation experiment

Here we will present results from the experiment described in section 5.1. To summarize, we generated a simulated haploid data set with average coverage 4.76. The truth file had 48469 SNPs, 11812 indels and no complex variants. The reads were mapped, realigned and applied to several variant callers, with and without realignment.

With such a low coverage, a significant number of both false positives and negatives is not only acceptable, it is also expected, no matter how good the algorithm is. This will be further explained in section 6.3.5.

Data from the different runs are collected in table 1 and commented in the following paragraphs. Note that all except one test use GATK IndelRealigner before variant calling.

Method	**Type**	**Sensitivity**	**Precision**
Raw UG	SNP	98.37	82.08
	indels	26.86	99.31
UG	SNP	98.36	84.00
	indels	34.20	98.87
HC	SNP	91.21	94.91
	indels	73.02	96.73
Dindel	SNP	0	0
	indels	76.71	92.55
BCFtools	SNP	20.97	85.19
	indels	14.03	22.68
VarScan 2	SNP	0.65	100

Table 1: Statistics on variant calling on haploid genome, 5x coverage, with different algorithms. UG = UnifiedGenotyper, HC = Haplotypecaller. Types defined in 4.6.2. All callers used realigned input, except "Raw UG". Dindel output was also leftaligned. Numbers generated with htslib vcfcheck simreads.type.vcf.gz sample.left.20.A.vcf.gz.

6.1.1 Variant generation

In section 5.1.3, it was described how to generate a representative variant set. First chromosome 20 in dbSNP was analysed, and statistics summarized in table 2 on the following page. Thereafter the variants files were generated, and the overlaps were found with vcf-compare as in table 3 on the next page. Only variant file A was used in the first haploid experiment, and even in a simplified form with only SNPs and simple indels.

6.1.2 VarScan 2 and bcftools

Both of these algorithms showed a striking underperfomance. To some degree, it might be due to a non-standard variant representation. VarScan uses elements like `-AG` or `+T` in the ALT column to denote deletions and insertions respectively, whereas bcftools

SNP frequency	0.866445
MNP frequency	0.002061
Deletion frequency	0.066676
Insertion frequency	0.064818
Total number of variants	1170635

Table 2: Statistics on variants in dbSNP from chromosome 20 as produced by generateVCFstatistics.py.

# Occurences	Which file(s)	(proportion of entire file)	
4951	A.vcf.gz (8.0%)		
4995	B.vcf.gz (8.0%)		
8839	A.vcf.gz (14.2%)	B.vcf.gz (14.2%)	
16272	B.vcf.gz (26.1%)	DB.vcf.gz (1.5%)	
16273	A.vcf.gz (26.1%)	DB.vcf.gz (1.5%)	
32193	A.vcf.gz (51.7%)	B.vcf.gz (51.7%)	DB.vcf.gz (2.9%)
1026628	DB.vcf.gz (94.1%)		

Table 3: Overlap between variant files generated by generateTestVCF.py. Numbers found using vcf-compare.

have REF and ALT columns with large overlaps. In other words, the indels are not written as compact as possible.

There were 5650 FP indels and 35 FP others in BCFtools, and 861 FP others in VarScan 2. But even if all these false positives are counted as true positives, both programs are still inferior to HaplotypeCaller. Thus we have not tried to convert the variant representation in this experiment.

Another possible explanation is that the coverage was too low for the algorithms to call variants. If this is the case, we should see an improvement when testing on a data set with higher coverage. This is especially true for VarScan 2 which uses a minimum coverage of 8 as default to call variants.

Because of the poor results, these algorithms have been omitted from the following discussion.

6.1.3 False positives

As can be observed in table 1 on the preceding page, there are several weaknesses in the algorithms. First, there is a high false positive rate for SNPs with UnifiedGenotyper (UG). Two substitutions are particularly over-reported, namely C→A and G→T with more than 2000 false positives each, which, for these substitutions, is more than the true number. Also HaplotypeCaller (HC) experiences many false positives.

One explanation can be the parameter *-stand_ emit_ conf* which was set to the value 10.0. On a phred scale, this translates to 90% confidence that the variant is correct. There are various filters in the GATK package designed to lower the false positive rate, which was not applied in this run, as described in section 4.5. However, these filters will not yield any new true positives, only remove false positives and, possibly, true positives.

Variant type	SNP		Indels	
UnifiedGenotyper	**sensitivity**	**precision**	**sensitivity**	**precision**
Unrealigned	98.37	82.08	26.86	99.31
GATK, knowns only	98.33	83.77	32.11	99.42
GATK, default	98.36	84.00	34.20	98.87
GATK, smith-waterman	97.78	84.10	34.22	98.85
SRMA	98.38	82.22	37.05	98.98
HaplotypeCaller	**sensitivity**	**precision**	**sensitivity**	**precision**
GATK, default	91.21	94.91	73.02	96.73
SRMA	91.21	94.96	73.00	96.68

Table 4: Statistics on two different variant callers after different realigners. Haploid genome, 5x coverage. Total number of SNPs and indels were 48469 and 11812 respectively.

6.1.4 Indel calling

A second observation is the bad quality of indel calling. In this case there is a huge difference between the UG with sensitivity 26.86%-34.20% and HC with sensitivity 73.01%. We also see, as expected, that using the IndelRealigner increases sensitivity on indel calling for the UG significantly, but still far from HC.

Dindel has the highest sensitivity of 76.71%, but a low precision, which is commented in section 6.2.1.

It is perhaps slightly surprising that there was no significant difference between "novel" variants, and those present in the database file supplied to IndelRealigner. When using only variants in both the truth and the database file, realigned UnifiedGenotyper found 610 of 36477 SNPs and 6483 of 9739 indels, corresponding to a sensitivity of 98.33% and 33.43% respectively. This suggests that the IndelRealigner did not use the database in an efficient way, and that the improvements came from the information provided by the mapped reads, which was the other source of information. Actually, these numbers suggest that indel sensitivity was slightly reduced because of the database, but this can be an error caused by bad variant representation in IndelRealigner.

6.1.5 Realigner comparison

One important question from section 6.1.4 is if the variant database was correctly applied to the IndelRealigner, since the effect was almost identical for the indels overlapping with the database and the indels which were not in the database.

Furthermore, is there an effect of using smith-waterman alignment as an additional model? What about SRMA? Realignment commands are as described in section 5.1.6, and to test their performance, all realigners are followed by variant calling using GATK UnifiedGenotyper.

From table 4 we see that using only known indels from a variant database is better than not using any realignment. Furthermore, the increase in sensitivity from unrealigned to knowns is only 5.25%, whereas from unrealigned to the default model has an increase of 7.34%. Thus the increase in sensitivity was 28.5% less for the "knowns only"-model compared to the default model. This is almost identical to the 25% novel rate incorporated in the variant file, which means that for variants already in the database, known (database) indels are almost as good as the default model.

Smith-waterman realignment had very little effect on the final result, and the results indicate that the default model is a good choice.

Despite its drawbacks, SRMA was also promising, as it had the highest sensitivity of all realigners. But the result is still far from the other variant callers; HaplotypeCaller and Dindel.

Also, combining SRMA with dindel reduced the sensitivity to 73.06%, though with a higher precision of 97.27%.

6.1.6 SRMA issues

SRMA has not been updated for a long time (see section 3.4), and there was also some run-time problems when we tried to use it. With the parameters as in section 5.1.6, SRMA quickly used all available memory, both with `-Xmx4g` and `-Xmx7g` and became unresponsive or crashed. However, when run on a shared computer with 128 GB of memory, it eventually managed to finish after using more than 29 GB of memory at its peak. As chromosome 20 is relatively small and the data set contained few reads, this is not promising for whole-genome high-coverage runs, and other parameters to SRMA should be considered if one decides to use it.

6.1.7 Realignment time usage

The time usage ranged from 134 seconds for "knowns only" and 149 for the default model, to 265 seconds for SW. But RealignerTargetCreator dominated GATK realignment, using 6 minutes. SRMA used 4 hours and 10 minutes, admittedly with a different setup, a computer with 128 GB of RAM and 64 cores, shared with many other users. One should, however, take the notes on timing in section 5.6.1 into account.

6.2 Variant representation

In section 5.2, we propose how to study the effect on different variant representations. How many of the false positives are in reality true positives, just with a different but equivalent variant representation? Can this be changed by variant correction?

This experiment was done on the small haploid data set, without MNPs and complex variations, thus still unrealistic. It is nonetheless interesting to investigate the scope of variant misrepresentations. The experiment can later be extended to the case of MNPs and other complex variants and to a larger, diploid genome.

Due to the poor results from VarScan 2 and BCFtools, and also due to their non-standard variant representation, these have been omitted from the experiment. That leaves UnifiedGenotyper with unrealigned, GATK-realigned and SRMA-realigned input, HaplotypeCaller and Dindel, with and without the use of GATK IndelRealigner.

Unlike in the other tables, we will here operate with false negatives and positives instead of sensitivity and precision, in order to see the absolute improvements through correction, but it is easy to convert using the formulas in section 2.8.2.

6.2.1 Impact on variant files

The correction algorithm was run on multiple variant files, and the results are summarized in table 5 on the facing page. The corrected data are shown, with improvements in parentheses. FN and FP are false negatives and positives respectively. Any variant reported as MNP or others was ignored, but this was mostly zero, in the worst case 5.

Caller	SNP (FN)	SNP (FP)	Indels (FN)	Indels(FP)
Unified, unrealigned	781 (-8)	10266 (-142)	8543 (-96)	7 (-15)
Unified, realigned	780 (-17)	8991 (-90)	7643 (-129)	9 (-37)
Unified, SRMA	768 (-15)	10214 (-100)	7311 (-125)	10 (-35)
Haplotype, realigned	4221 (-38)	2233 (-136)	2858 (-329)	131 (-161)
Haplotype, SRMA	4223 (-38)	2211 (-136)	2860 (-329)	135 (-161)
Dindel, unrealigned	48433 (-36)	0 (0)	3007 (-132)	152 (-91)
Dindel, realigned	48433 (-36)	0 (0)	2633 (-134)	187 (-824)

Table 5: Summary of different variant callers. Haploid genome, 5x coverage. Variant corrected, gain from uncorrected in parentheses.

There are several important observations. One is that there is generally a great reduction in false positive SNPs and false negative indels. In other words, what is written as a set of indels in the truth file, and used for generation of haplotypes, can be written differently using SNPs.

The exceptionally high number of reduced false positives in Dindel is due to a high number of duplicates in the variant file. Deeper study revealed 738 duplicates.

Where the reduction in false positives is larger than the reduction in false negatives, variants callers are not using the shortest possible representation. But there are also examples of combinations from dbSNP that could be written more easily, using fewer variants.

6.2.2 Variant file equivalence

As the tool is self-made and experimental, we wanted to verify the equivalence of the variant files, by generating the haplotypes and comparing them with unix diff. This test was done with the SRMA/UnifiedGenotyper and Dindel variant files. Due to filtering by GATK FastaAlternateReferenceMaker, the variant files had to be modified, removing the "LowQual" identifier from UnifiedGenotyper and similar identifiers from the variant file from Dindel.

In case of SRMA there was a handful of lines differing, all related to positions with multiple variants. In this case, FastaAlternateReferenceMaker will choose one at random, which gives a high probability of stray mismatches.

With dindel the files were identical. It was therefore assumed that the algorithm was correctly implemented.

6.3 Simple haplotype, better coverage

In the experiment with the second data set, which is described in section 5.3, the number of reads was quadrupled, though still haploid. This was done to verify if the results from the previous experiment were only due to bad coverage. The commands were exactly the same, except that the number of reads were increased to $12 * 10^6$. In addition we wanted to investigate the relation between coverage, measured coverage and variant calling accuracy.

6.3.1 Variant caller summary

The sensitivity and precision of the different variant callers are listed in table 6 on the next page, and the absolute numbers and effect of variant correction are listed in

Caller	SNP (sens.)	SNP (pre.)	Indels (sens.)	Indels(pre.)
UnifiedGenotyper	99.65	94.43	88.29	99.39
HaplotypeCaller	99.45	96.54	89.83	96.78
Dindel	0.08	100	90.79	97.79
VarScan 2	17.22	99.92	0	0
BCFtools	36.36	99.35	85.73	95.88

Table 6: Summary of sensitivity and precision of different variant callers. Haploid genome, 19x coverage, variant corrected.

Caller	SNP (FN)	SNP (FP)	Indels (FN)	Indels(FP)
UnifiedGenotyper	169 (-53)	2849 (-131)	1383 (-272)	64 (-156)
HaplotypeCaller	269 (-76)	1727 (-177)	1201 (-467)	353 (-239)
Dindel	48428 (-1)	0 (0)	1151 (-159)	241 (-110)
VarScan 2	40125	7	11812	9750
BCFtools	30845 (-40)	115 (0)	1686 (-8235)	435 (-8128)

Table 7: Summary of different variant callers and the effect of variant correction. Gain from uncorrected in parentheses. Haploid genome, 19x coverage.

table 7. All callers used reads realigned with GATK IndelRealigner. The truth file was leftaligned and all trial files were variant corrected, except VarScan 2, which had an unconventional variant format. Total number of SNPs and indels was 48469 and 11812, respectively, like in the previous experiment.

From the table, we see that VarScan 2 and BCFtools are still mediocre. This is also true if all FP indels in VarScan 2 in reality are just misrepresentation due to the unconventional formatting. However, it can seem that both algorithms are less aggressive, as the number of false positives is low. But with such a low sensitivity, the results are still bad. UnifiedGenotyper on unrealigned input was also tested, and found to be strictly worse than with realigned input, similar to the previous experiment.

6.3.2 Sensitivity and precision with different indel size

Because htslib vcfcheck reports detailed statistics about which types of variants are in the different sets, it is possible to report individual data for all indels with a particular size, negative for deletions and positive for insertions. This has been done in figures 3 on the facing page and 4 on page 60. Indels longer than 20 bp are omitted due to low sample size.

From the figure we see a sharp drop in sensitivity at indel size 5. This is in particular true for UnifiedGenotyper with unrealigned input. But also realigned UG is very careful and calls very few variants. HaplotypeCaller finds a few more, but at the cost of worse precision. Dindel both has a high sensitivity and precision on longer indels.

It is clear from the figures that deletions are much easier to call than insertions. This is not very surprising. Insertions are defined as a position and a completely new sequence that is to be inserted, whereas deletions only are defined as a set of consecutive positions that are to be removed.

6.3.3 Gain from variant correction

Table 6 also contains the gain from the uncorrected variant files. It is clear that BCFtools use an uncommon representation of indels, as was suspected.

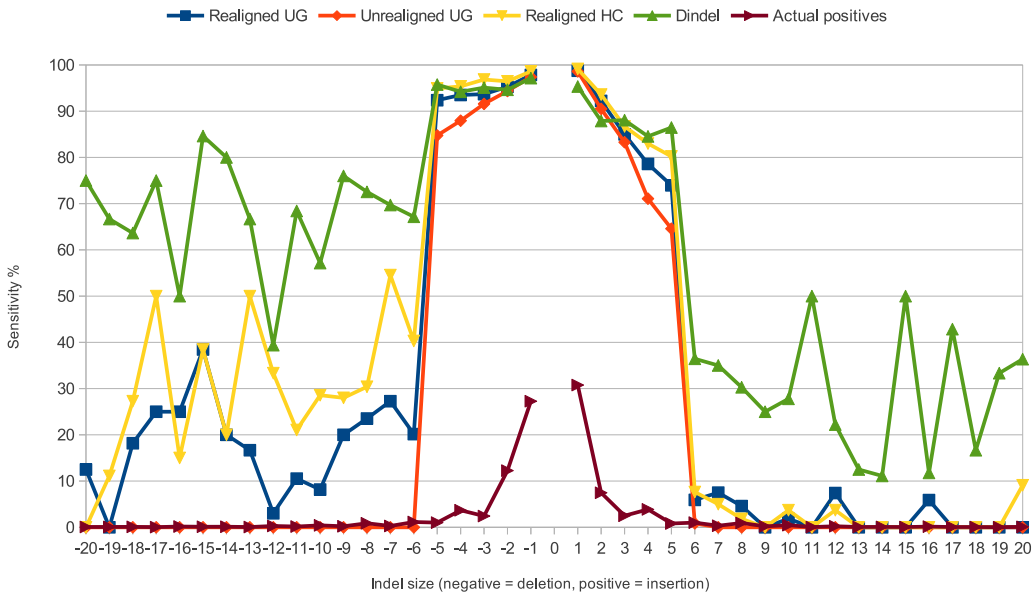

Figure 3: Indel sensitivity with different variant callers on haploid genome, 19x coverage, variant corrected. UG = UnifiedGenotyper, HC = HaplotypeCaller. The figure also shows the number of indels with a particular size as a share of the total number of indels.

We also see that HaplotypeCaller gained the most of the remaining variant callers. In other words, these variants differed the most from those in dbSNP. One may speculate that this is because HaplotypeCaller was not used to generate dbSNP, as it is a very new algorithm. This theory was not studied further though.

A consequence is, however, that we see that HaplotypeCaller is more sensitive than UnifiedGenotyper at detecting indels, a fact hidden without variant correction.

The fewest improvements were gained with Dindel. There are two possible explanations. Either the results were good in the first place, after all, it is the variant caller with the most found indels, or it is because the variant corrector requires an exact equivalence, but this may in some cases only be possible if SNPs are replaced as well as indels.

The occurences of nonstandard variant representation in dbSNP strengthen the second hypothesis.

6.3.4 Variants and coverage

In the previous experiments, we have summarized data from the entire chromosome. However, as mentioned before, the coverage is variable. In particular it may be very low or zero in some areas. We have calculated the coverage using samtools as described in section 5.3.2.

To avoid the problems described in section 4.8, the reported coverage is not used directly. Rather, it may be increased if it is higher on both sides of the position, up to 10 bp away.

From the variant file we get the leftmost position of the variant, and this is matched with the modified reported coverage. The next step is to split the truth variant file

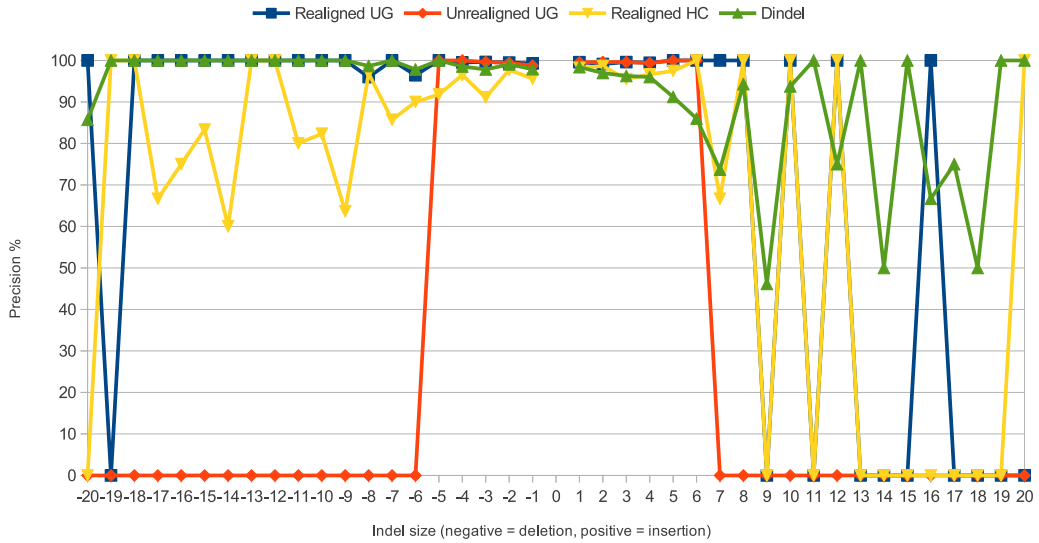

Figure 4: Indel precision with different variant callers on haploid genome, 19x coverage, variant corrected. UG = UnifiedGenotyper, HC = HaplotypeCaller. Where no indels were called, precision was defined to be 0.

based on reported coverage. In order to keep the number of variant files manageable, some of them are merged.

To see if coverage had an effect on variant calling, we studied the overlap between the split truth files and the corrected variant file from UnifiedGenotyper, realigned input. The results can be seen in table 8 on the next page. False positives are omitted to avoid the need of splitting the other variant file as well, and also because the numbers are less trustworthy due to the lack of a filtering step. Only SNPs and indels are reported, as the other variant types are too infrequent.

From table 8 on the facing page, we see some interesting facts. Naturally, the sensitivity is very low in the areas with low coverage. This is particularly true for indels, which also have a low overall sensitivity.

On the other hand, it is very interesting to look at the expected number of variants. This is calculated as the probability of a certain coverage, based on the poisson assumption, multiplied by the total number of variants. As we can see, the number of variants in low-coverage regions is much higher than expected. Because the truth file is generated independent of any reads, this implies that variants cause low coverage, and not the other way around.

There are two possible explanations. One is that variants cause a low reported coverage, for instance due to deletions. The other explanation is that variants cause reads to be mapped to the wrong position. From the data we cannot exclude the first theory. However, that high-coverage variants are also slightly overrepresented, indicates bad mapping. This is because insertions would not increase reported coverage, and thus the increased coverage must be caused by erroneous mapping in that particular area.

A third option is that it happened by chance, but if we assume a binomial distribution on the number of variants with coverage above 25, with a success probability of 0.08114 and a sample size of 60281 variants, we get a mean of 4891 and a standard deviation of 67. The probability of obtaining a result 6.68 standard deviations above

	SNPs		Indels		Num. of variants	
Coverage	**FN**	**TP**	**FN**	**TP**	**Total**	**Expected**
0	6	0	33	0	39	$2 * 10^{-4}$
1	8	4	63	0	75	$5 * 10^{-3}$
2	9	6	137	0	152	0.05
3	5	6	161	0	172	0.32
4	6	18	183	1	208	1.5
5	6	21	147	17	191	5.9
6	2	26	135	26	189	18.8
7	3	33	101	60	197	52
8	7	107	66	75	255	124
9	2	186	78	87	353	266
10	8	304	50	155	517	511
11-15	32	7661	147	2072	9912	11088
16-20	58	20231	61	4254	24604	25737
21-25	11	15202	14	2885	18112	17584
26+	6	4529	7	797	5339	4891

Table 8: Variants statistics split on coverage. Haploid genome, 19x coverage. FN/TP calculated using realigned, variant corrected UnifiedGenotyper.

the mean, only by chance, is extremely low, about 10^{-11}.

Whether the high number of false negatives are caused by mapping errors or is a consequence of the low coverage, is not investigated further.

6.3.5 Poisson assumption on coverage

In table 8 we assumed poisson distributed coverage. To verify this claim, we ran the tools described in section 5.3.2 on both unrealigned and realigned mapped reads. The coverage frequency can be seen in figure 5 on the following page for the 5x-coverage data in the previous experiment and in figure 6 on the next page for the 19x-coverage data in this experiment. We also see the probability of a position having a given coverage, assuming poisson distribution with the average coverage as a parameter.

The first observation is that the poisson assumption seem to hold. The only exceptions are for high and low coverage.

In the 5x-coverage case, we would expect almost 1% positions with zero coverage, but the reported coverage is only about one hundredth. This is probably due to behaviour of samtools depth as described in section 4.8.1.

In the 19x-coverage case, the expected number of positions with zero coverage is less than zero, and there are also very few expected positions with low coverage. Deviations that are small in absolute number can thus still be clearly visible in the graph.

6.4 Diploid genome

As described in section 5.4, a diploid genome with paired-ended reads was simulated to get a more realistic experiment. The GATK variant callers and Dindel were compared, both for homozygous and for heterozygous variants. In addition, the variant files were merged to look for variants that were unique to a certain variant caller.

Two partly overlapping variant files were drawn according to the frequencies found in section 2.4.4 and table 3 on page 54. The number of variants in the truth file is 66785

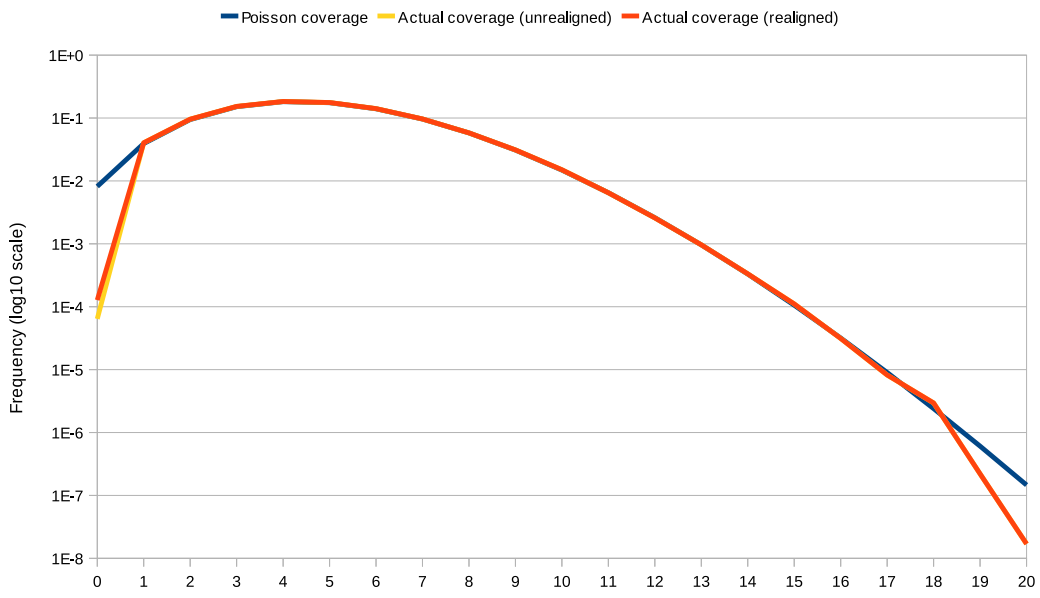

Figure 5: Frequency of positions with a given coverage, along with the expected number of positions assuming poisson distribution. Haploid genome, 5x coverage.

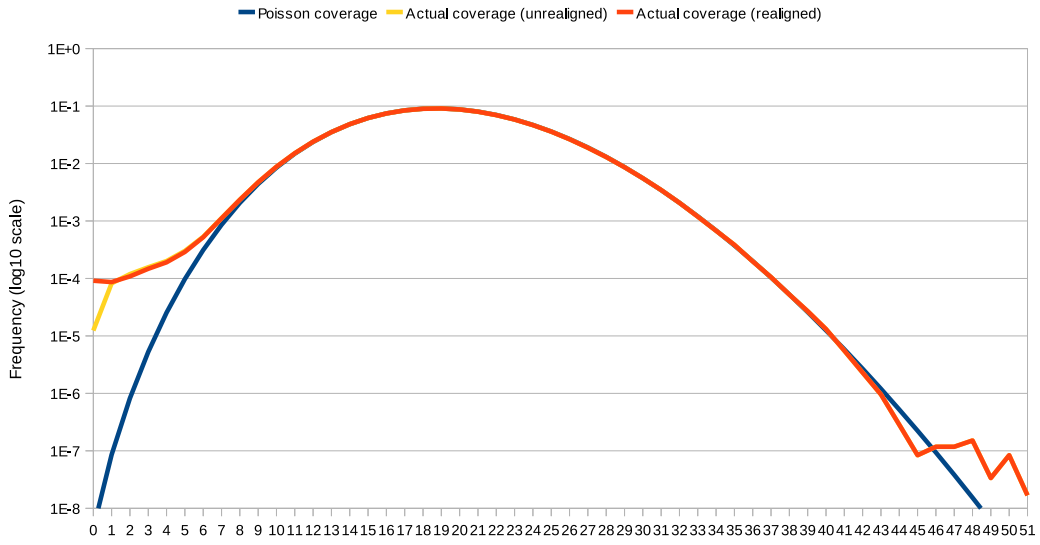

Figure 6: Frequency of positions with a given coverage, along with the expected number of positions assuming poisson distribution. Haploid genome, 19x coverage.

SNPs, 1101 MNPs, 14911 indels and 519 others. If we only look at the homozygous variant present in both truth files, the corresponding numbers are 29980 SNPs, 320 MNPs, 10236 indels and 341 others.

The results are summarized in table 9 on the next page. All numbers are using variant corrected data on realigned input.

6.4.1 Variant caller analysis

There are several interesting observations from the three first blocks of table 9 on the following page. HaplotypeCaller is clearly standing out as the best variant caller for indels. This is particularly true for heterozygous indels which have a sensitivity similar to the homozygous indels. Dindel, which performed well on the haploid genome, is now falling behind.

When looking at SNPs, UnifiedGenotyper is the best, though this may be due to a higher aggressiveness, as the low precision indicates.

Also MNPs were found, with a slightly lower sensitivity as the indel calling, but with high precision. The high precision is only because of variant correction, as none of the callers had any true positive MNPs originally, except dindel which had one true positive out of 1101.

6.4.2 Advanced analysis

In the last 4 blocks of table 9 on the next page, we have summarized what happens if we combine the results from different variant callers. This analysis can tell us whether there are unique variants from all callers, or if some algorithms are strictly inferior. Note that variants were corrected after the merges.

One interesting observation is the indel precision with HaplotypeCaller and Dindel, which is higher than the precision with Dindel alone. This suggests that some of the FP from Dindel are corrected when variants from HaplotypeCaller are made available. Same observation can be made from the SNP precision in Dindel and UnifiedGenotyper. Note that FP is related, but not equivalent to precision, but that the arguments would still hold in this case if we used FP instead of precision.

There are also peculiar observations. The sensitivity of "others" in HaplotypeCaller *drops* when the results are merged with UnifiedGenotyper. There are several examples like that, and although the differences are small, this is an error. A subset of a variant file should never have higher sensitivity. This error may come from incomplete merging, differences in variant correction or incorrect counting.

There were some issues during merging, vcf-merge that was used warned about positions appearing twice in dindel and in particular in UnifiedGenotyper. Also dindel produced a non-standard vcf file with headers that had to be modified prior to merging. The alternative, vcf-isec is more silent and does not emit warnings, only errors.

The combined results confirm several of the findings from the single results. HaplotypeCaller is the best indel caller; even when combining UnifiedGenotyper and Dindel, the result is far inferior. Similarly is UnifiedGenotyper the best SNP caller, but this may be due to aggressiveness and a lower precision.

Furthermore, Dindel is seemingly better at indels than UnifiedGenotyper. HaplotypeCaller merged with Dindel has a precision of 99.64, compared to 99.59 when HaplotypeCaller is merged with UnifiedGenotyper. However, this is a very weak conclusion, but still surprising given the low indel sensitivity in Dindel compared to UnifiedGenotyper, 95.34 compared to 96.88. This also suggests that different representation is a

Caller	Truth Type	All variants		homozygous variants
		Sensitivity	Precision	Sensitivity
UG	SNP	99.89	98.78	99.93
	MNP	96.09	100	97.81
	indels	96.88	96.04	98.25
	others	44.70	97.89	46.04
HC	SNP	99.58	99.54	99.62
	MNP	97.91	99.63	99.06
	indels	99.07	97.61	99.17
	others	51.06	99.25	52.49
Dindel	SNP	0.15	100	0.12
	MNP	2.27	86.21	2.81
	indels	95.34	95.36	95.90
	others	15.80	80.39	14.96
HC+Dindel	SNP	99.59	99.54	99.62
	MNP	97.91	99.35	99.06
	indels	99.64	96.59	99.72
	others	51.45	89.30	53.08
HC+UG	SNP	99.95	98.54	99.96
	MNP	98.37	98.81	99.69
	indels	99.59	96.12	99.70
	others	50.48	91.93	51.91
Dindel+UG	SNP	99.91	98.86	99.94
	MNP	96.91	98.80	98.75
	indels	98.53	95.51	98.92
	others	46.24	86.02	46.92
All 3	SNP	99.95	98.55	99.96
	MNP	98.37	98.28	99.69
	indels	99.65	95.62	99.74
	others	50.48	84.52	51.91

Table 9: Summary of different variant callers and their union. UG = UnifiedGenotyper, HC = HaplotypeCaller. Variant correction is applied after merging. Diploid genome, 64x coverage.

bigger problem for Dindel, which cannot correct all indels from the lack of SNPs.

6.4.3 Homozygous and heterozygous variants

In table 9 there is a column for sensitivity of homozygous variants. In general, homozygous variants are easier to find, since the expected number of reads containing the variant is doubled. In the table we see that the sensitivity is higher for homozygous variants and, consequently, lower for heterozygous variants. For HC and Dindel, the difference is quite low, for UG it is slightly higher.

6.4.4 Coverage and poisson

A total of 10^7 reads were drawn from each haplotype, each read with a length of $2 \cdot 101$ base pairs. This gives an average coverage of $2 \cdot 10^7 \cdot 2 \cdot 101 / 63025520 = 64.10$. Average coverage on each haplotype is half of this number.

In figure 7 we see the probability of a position having a given coverage, assuming poisson distribution with average 64.1. On the same plot we see the actual coverage distribution on the 59505502 mapped positions. The analysis is similar to that in section 6.3.5.

The line representing poisson coverage extends far below the plot area. The lowest Y-position on the actual coverage line represents exactly one genome position.

The plot reveals an excellent match for coverages between 35 and 100, strengthening the poisson assumption also for the diploid genome. We see a rise in frequency for low coverages, which may be caused the method of counting coverage at deletions. But there is also a relatively high number of positions with a high coverage. Even though the absolute frequency is still low, this confirms the results from section 6.3.4, that the mapping/realignment algorithm is not mapping all reads to the correct location.

If one looks closely at the plot, we see that the most common coverage is 63, followed by 62 and 64. The poisson line seems to be slightly further to the right. And the numbers confirm the trend, the average coverage is 63.21 across all mapped positions. This means that there were some reads that could not be mapped. Note that unmapped positions, for instance long sections with "N"s are ignored when calculating the coverage.

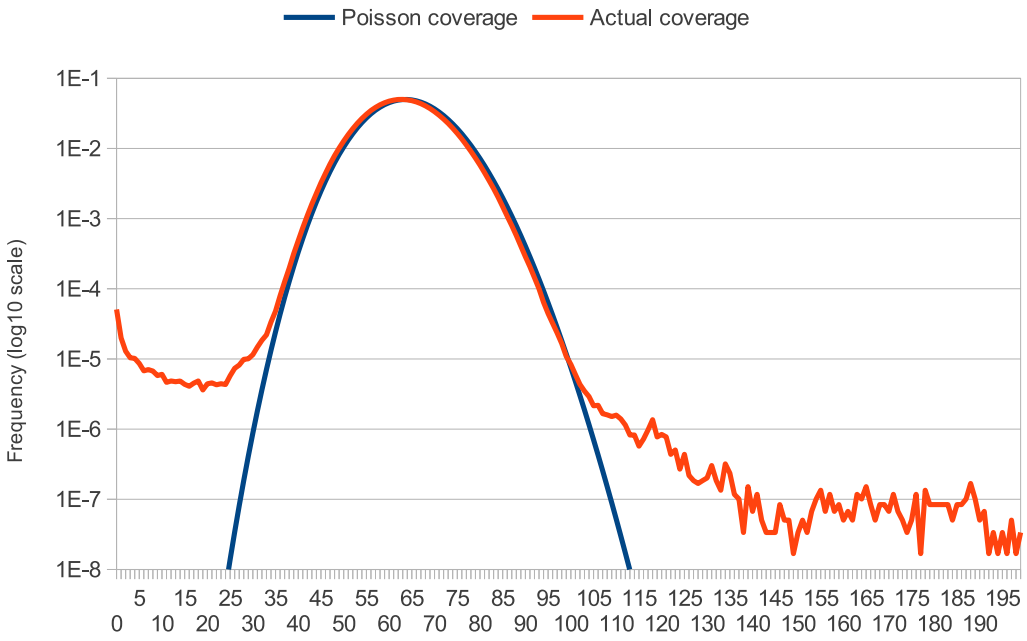

Figure 7: Frequency of positions with a given coverage, along with the expected number of positions assuming poisson distribution. Diploid genome, 64x coverage.

To investigate the cause of high coverage, some of the positions with high coverage were studied manually using IGV. A screen shot can be seen in figure 8 on page 67. All positions with coverage higher than 169 are from this single region. The image is cropped to fit on one page, but this high-coverage region spans about 300 bp.

All the white rows, and there are many of those outside the figure as well, represent mappings with the parameter "mapping quality" set to zero. And many of the remaining rows also have low quality. This means that the mapper knows that many of the reads are of poor quality.

There were no variants in the affected area, but close observations of the reference genome reveals 19 repetitions of a sequence with 59 bp. This results in the region with extremely high coverage, but also, as can be seen on the top of the image, a region with an extremely low coverage.

The region with the second highest coverage, around base pair 58985000 in chromosome 20, confirms the observation with extreme coverages in repetitive regions.

6.4.5 Variant file merging

It was expected that different methods for merging the same variant files would produce the same result. However, the two programs in vcftools, vcf-isec and vcf-merge produced files with a different variant set.

Using the parameter `-n +1`, which according to the documentation of vcf-isec yields all variants from at least one file, or a standard union, there were variants unique output files from both vcf-isec and vcf-merge.

The same was the case with the parameter `-n +2`, or a standard intersection. In this case, we would expect the output from vcf-isec to be a proper subset of vcf-merge, using only the homozygous variants, but even here there were unique variants in both output files.

However, vcf-isec with parameter `-n +2` was a proper subset of vcf-isec with parameter `-n +1`, therefore vcf-isec was used for all variant file merging and intersecting.

What is even more worrying is that vcf-isec, without warning, drops variant from the union. That is, htslib vcfcheck reports variants in a variant file that is not present in the union of that exact file and another file.

vcf-merge is more verbose than vcf-isec and warns for instance when multiple variants appear at the same position.

6.4.6 Timing

The different algorithms were very different in time usage, subject to the limitations mentioned in section 5.6.1. The numbers are given in table 10 on page 68.

For each algorithm, it is marked if the program was run more than once, with the number of required repetitions. In that case, the time value is an average.

From the table, we see that the computationally hardest programs were read generation, Dindel and HaplotypeCaller. Read generation is not relevant for biological samples, which leaves Dindel and HaplotypeCaller.

It is not possible to predict the time usage on full-genome calling from these data alone. Time complexity may depend on number of reads, length of reference and other factors.

6.5 Variant recalibration

In section 5.5 we described how to generate more detailed description of the variant call performance with different aggressiveness levels. This was achieved by ordering variant files after variant calling quality, set a different cutoff value than what was used in the parameter given to the variant caller, and re-calculate the sensitivity and precision for the caller.

Both the medium-sized and large data set was investigated.

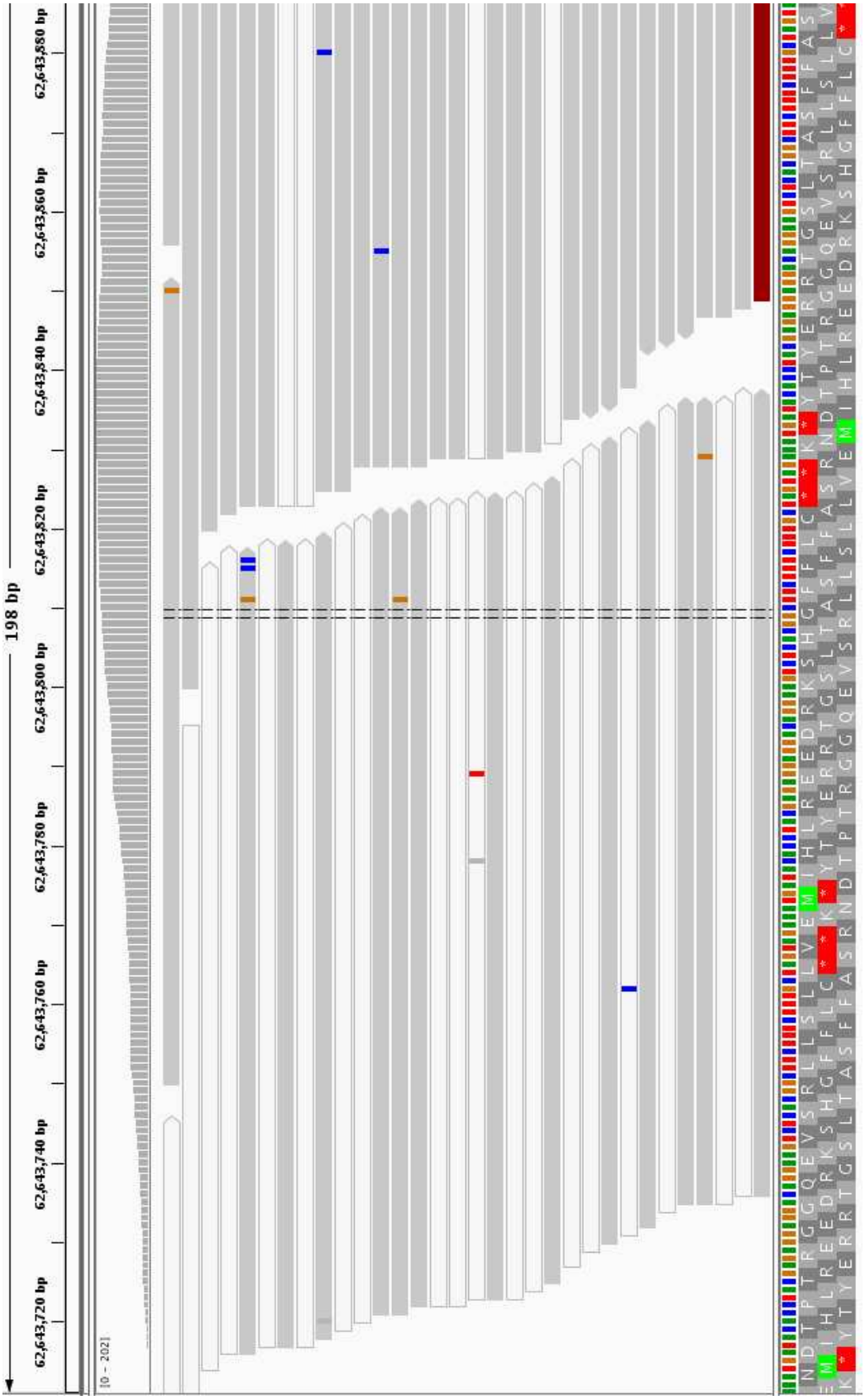

Figure 8: Screen shot of IGV, diploid genome, 64x coverage, repetitive region.

Generate test VCF files	18 seconds
Leftalign variant files	11 seconds (two times)
Create alternative haplotype	9 minutes (two times)
Generate 10^7 reads	9 hours (two times)
Mapping, aln part	17 minutes (four times)
Mapping, sampe part	11 minutes (two times)
Convert from SAM to BAM	13 minutes (two times)
Merge BAM files	24 minutes
Add read groups+sort+index	26 minutes
GATK Realignment, targetcreator	18 minutes
GATK Realignment, indelrealigner	28 minutes
Variant calling, UnifiedGenotyper (8 threads)	18 minutes
Variant calling, Dindel, getCIGARindels	2 minutes
Variant calling, Dindel, makeWindows	4 seconds
Variant calling, Dindel, indel analysis	23 minutes (48 times)
Variant calling, Dindel, mergeOutput	8 seconds
Variant calling, HaplotypeCaller	47 hours

Table 10: Approximate time usage on diploid data set, coverage 64.

6.5.1 19x coverage experiment

An analysis was first performed on the haploid, 19x coverage genome. The results can be seen in figures 9 on the facing page and 10 on page 70, and also compared with the original table 6 on page 58.

In the figures, there can either be a drop in precision, when there were false positives, or the graph will go to the right and up with a shape like $-x^{-1}$ in the case of true positives.

In the case of SNPs, we see a very good performance by UnifiedGenotyper, except for the very last SNPs that it aggressively tries to find. This drop in precision masks the fact that for a given sensitivity level, UnifiedGenotyper is always better than HaplotypeCaller. The only exception is for low sensitivity levels, where stray false negatives can have a large effect on precision.

Deeper study into the numbers reveals that at 17.22% and 36.36% coverage, UG had a precision of 99.95% and 99.97%, respectively. It was thus better than both VarScan 2 and BCFtools.

With indels, the differences were harder to spot. It looks as if UG is superior here as well, and that dindel slightly outperforms HC. But, because UG is less aggressive, its final sensitivity is lower. Using a more aggressive threshold in UG may reveal that it is strictly better than the alternatives, but that is impossible to say from these data.

6.5.2 Error sources

The script used was self-written, using a different way of handling difficult variant representations. The total number of SNPs found was correct, 48469, but only 11801 indels was found in total, 11 less than what was reported by htslib vcfcheck.

The script also splits lines with alternative variants, generating more variants at the same position. All variants were checked for an identical variant in the truth file. The final sensitivity and precision reported was 99.65% and 94.37% for UG SNPs, 99.39% and 96.53% for HC SNPs, 88.37% and 99.39% for UG indels, 90.34% and 97.74% for

dindel indels and 89.92% and 96.69% for HC indels. These numbers differ slightly from what is reported in table 6 on page 58, but not so much as to change the relative order of the callers.

Light manipulation of the quality scores was also performed; when variants were corrected, the lowest quality score of all source variants was used in the corrected file for all replacements.

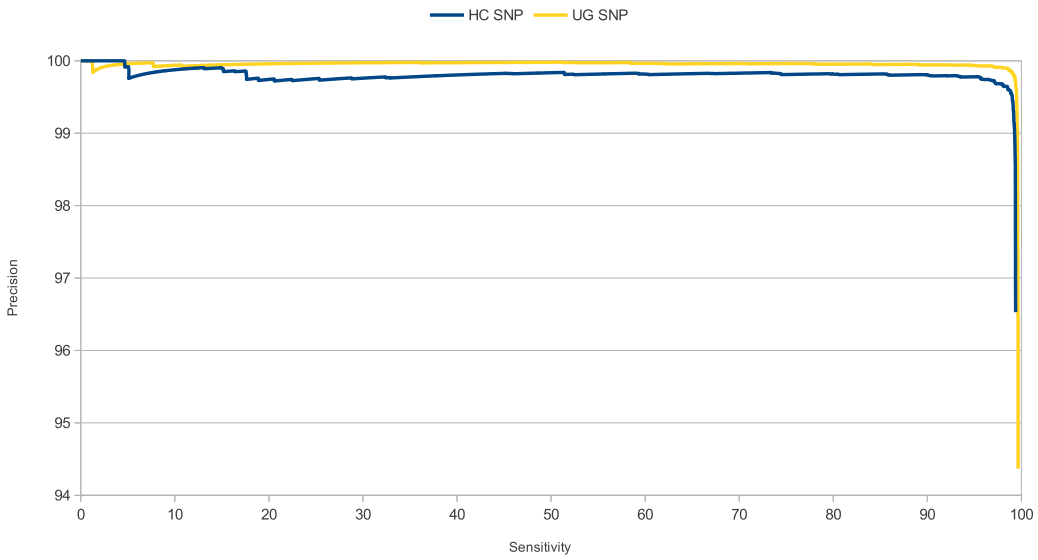

Figure 9: SNP sensitivity and precision with HaplotypeCaller and UnifiedGenotyper using different quality thresholds. Haploid genome, 19x coverage.

6.5.3 64x coverage experiment

The same experiment was done on the diploid data set, on the three main variant callers. The result can be seen in figures 11 on page 71 and 12 on page 71, and compared with the data in table 9 on page 64.

For SNPs, the precision of HC was generally much higher than UG, though with a lower final sensitivity. HC has a maximum sensitivity of 99.58% and a corresponding precision of 99.53%. At this level of sensitivity, UG had a precision of 99.59%.

This means that HC and UG are about equally good at calling SNPs at high sensitivity, but that HC is better at lower sensitivity levels.

Another observation is that there were several false positive SNPs with UG that had a very high quality score. It was not investigated further, and the errors may be explained by differences in, for instance, variant representations. It can also be that a more advanced recalibration is beneficial, for instance the GATK tool described in section 4.5.

For indels, the results clearly indicate that HC is the best indel caller. None of the algorithms were particularly aggressive, as there are no sudden drops in precision at high sensitivity.

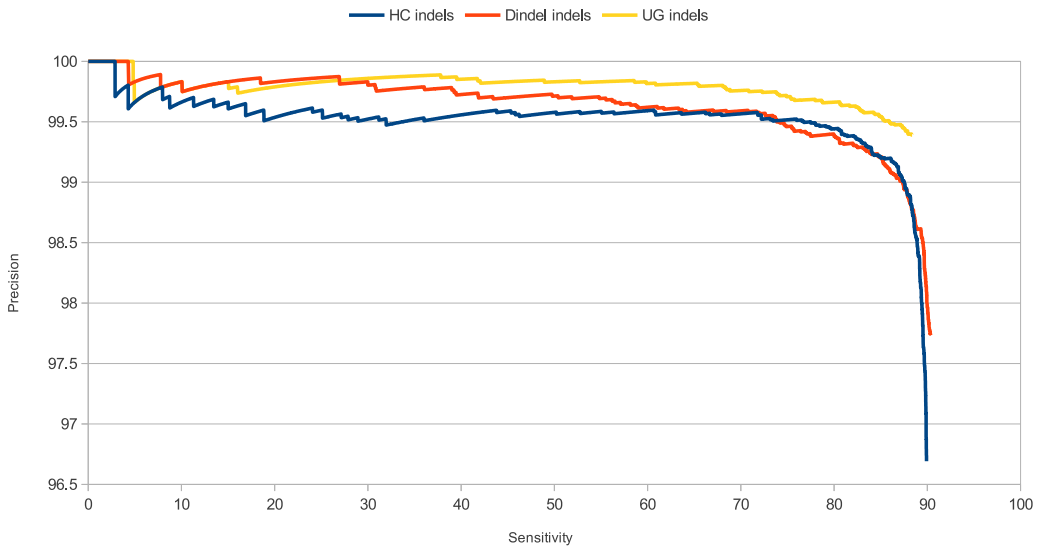

Figure 10: Indel sensitivity and precision with HaplotypeCaller, UnifiedGenotyper and Dindel using different quality thresholds. Haploid genome, 19x coverage.

6.5.4 Error sources

The same error sources apply to the diploid genome experiment. Even though the SNP count remained the same at 66785, the number of truth indels was significantly reduced from 14911 to 13965.

Also, as described in section 6.4.2, there are problems with variant representation, in particular for dindel. This must be taken into account when comparing variant callers.

The merged variant files were not considered, as there is no obvious strategy of determining variant quality scores when merging results from different variant callers.

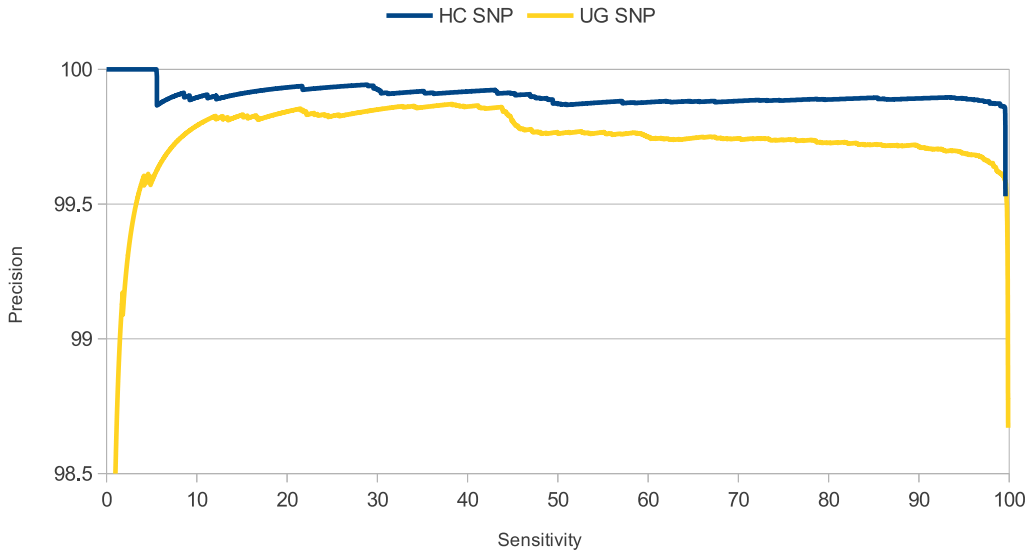

Figure 11: SNP sensitivity and precision with HaplotypeCaller and UnifiedGenotyper using different quality thresholds. Diploid genome, 64x coverage.

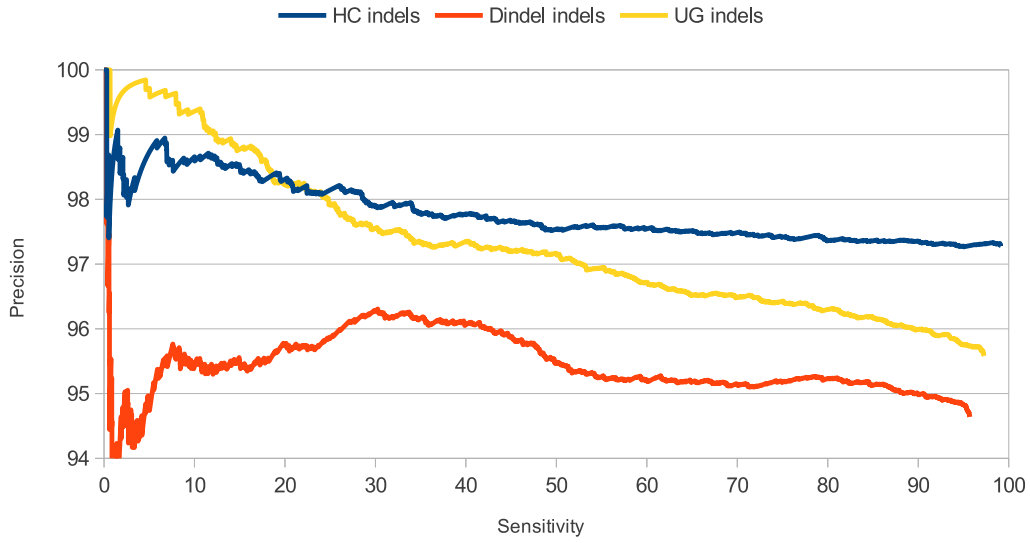

Figure 12: Indel sensitivity and precision with HaplotypeCaller, UnifiedGenotyper and Dindel using different quality thresholds. Diploid genome, 64x coverage.

7 Discussion

In this section we will try to summarize the results, draw conclusions and point out the weak points from the experiments that require further study.

7.1 Variant sources and representation

Using dbSNP as a source of variants caused some problems. Not all variants were minimal or left-aligned. SNPs may appear in dbSNP as MNPs, and indels may appear as complex variants. This can somehow be corrected by pre-filtering bad variants or rewriting variant representations. Results showed that many complex variants like indels in dbSNP could be rewritten using SNPs.

An alterantive is to choose another source of variants with higher quality, for instance 1000 genomes, or to filter the variants through a variant caller as outlined in section 4.7.3.

The alternative to good variant sources is to apply variant correction or advanced comparison. This can be done after variant calling.

When generating variant files for simulated reads, it is essential that the variants are formatted in a way that all programs can understand and use. This includes programs for generating alternative haplotypes and programs for variant manipulation and comparison. If necessary, the variant files must be converted to an equivalent representation.

No matter which variant database is used, we still have the problem with bias. The databases only contain variants which have been found at least once, that is, the "easy" variants. This applies particularly to callers frequently used to generate variants found in dbSNP.

Using biological instead of simulated reads does not remove the bias completely, as a variant caller is needed to generate the truth file.

7.2 Mapping and realignment

BWA was the only mapper that was tested. Nevertheless, it is not error-free. The coverage can drop when variants, in particular indels, are present, making good variant prediction hard. The coverage may also be much too high, when reads are mapped to wrong areas. In addition, repetitive regions may cause erroneous mapping.

These issues call for good realignment, and GATK IndelRealigner provides a significant improvement, but it could be even better. As the time used on realignment, admittedly only on one small chromosome, was less than the time used on mapping, we expect that there is room for more complex algorithms.

SRMA realignment was buggy, slow and not very different from GATK in performance. This may be the reason it is out of active development.

An interesting line for further study is to annotate the raw reads with correct position in the simulator. With both the correct and mapped position, it is possible to calculate the distribution of mapping errors accurately. Together with a list of variants, we may also find out what causes these mapping errors and use this information to improve realigners.

7.3 Variant calling

The best variant callers were found to be GATK HaplotypeCaller (HC) and UnifiedGenotyper (UG), with Dindel being interesting as a supplementary algorithm.

UG was best on the haploid genome, both for SNPs and indels. UG was also very fast, compared to HC and Dindel. On the diploid genome, HC was very good on indels and complex variants. Dindel performed reasonably good on indels, particularly in combination with HC. The results are not conclusive on diploid SNP calling. HC seem to be more precise, but at very high levels of sensitivity, UG may be better.

Compared to these variant callers, bcftools and VarScan 2 were both inferior, and did not conform to file standards and conventions either. The average coverage of 19 is not low enough to justify the poor results. Nevertheless, VarScan 2 may be appropriate for other problems.

GATK recommends post-filtering which should improve precision. Simple filtering is also possible by adjusting the quality score threshold. Filtering had a great effect on calling precision and sensitivity, but which approach that is most appropriate can be studied further.

7.4 Variant correction and manipulation

Comparison and manipulation of variant files is an area that calls for extensive improvements. A good idea would be to improve the functionality and algorithm of our variant correction tool, to increase speed and possibly sensitivity. Then that algorithm could be implemented as an option in htslib vcfcheck or vcf-compare, which would then be able to compare variant files for equivalence, not just equality.

Furthermore, there can be improvements in merging and/or intersection of variant files. Much can be improved simply by writing proper documentation that explains the problematic examples, for instance what happens with overlapping variants.

The quality of variant files from different callers varied. vcf-merge complained about duplicates in UG and Dindel, but not HC, indicating a higher quality of variants from HC. VarScan 2 and bcftools had even more issues.

It is also interesting to follow the developement of htslib which is in the process of implementing its own intersection and merge algorithm.

7.5 Conclusion

Read simulation is a good way of obtaining valuable data on variant calling. It can reveal weaknesses and strengths in multiple steps in the pipeline; mapping, realignment, calling and filtering. It is crucial, however, that read generation is done correctly, to avoid introducing unknown errors and biases.

There are also issues with comparison and validation of variants from simulated reads, and these algorithms must be carefully constructed, to avoid variants to be erroneously reported as false positives.

GATK proved to be a good alternative for variant calling, but there is still work to be done, especially in realignment or in mapping.

8 Bibliography

References

[1] Cornelis A Albers, Gerton Lunter, Daniel G MacArthur, Gilean McVean, Willem H Ouwehand, and Richard Durbin. Dindel: accurate indel calls from short-read data. *Genome Research*, 21(6):961–973, June 2011.

[2] Adrian Bird. Perceptions of epigenetics. *Nature*, 447(7143):396–398, May 2007.

[3] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital Equipment Corporation, 1994.

[4] Markus Christmann, Maja T. Tomicic, Wynand P. Roos, and Bernd Kaina. Mechanisms of human DNA repair: an update. *Toxicology*, 193(1-2):3–34, November 2003.

[5] Peter J. A. Cock, Christopher J. Fields, Naohisa Goto, Michael L. Heuer, and Peter M. Rice. The sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. *Nucleic Acids Research*, 38(6):1767–1771, April 2010.

[6] David W. Collins and Thomas H. Jukes. Rates of transition and transversion in coding sequences since the human-rodent divergence. *Genomics*, 20(3):386–396, April 1994.

[7] The 1000 Genomes Project Consortium. A map of human genome variation from population-scale sequencing. *Nature*, 467(7319):1061–1073, October 2010.

[8] Adrian V Dalca and Michael Brudno. Genome variation discovery with high-throughput sequencing data. *Briefings in Bioinformatics*, 11(1):3–14, January 2010.

[9] Petr Danecek, Adam Auton, Goncalo Abecasis, Cornelis A Albers, Eric Banks, Mark A DePristo, Robert E Handsaker, Gerton Lunter, Gabor T Marth, Stephen T Sherry, Gilean McVean, and Richard Durbin. The variant call format and VCFtools. *Bioinformatics (Oxford, England)*, 27(15):2156–2158, August 2011. PMID: 21653522.

[10] Mark A DePristo, Eric Banks, Ryan Poplin, Kiran V Garimella, Jared R Maguire, Christopher Hartl, Anthony A Philippakis, Guillermo del Angel, Manuel A Rivas, Matt Hanna, Aaron McKenna, Tim J Fennell, Andrew M Kernytsky, Andrey Y Sivachenko, Kristian Cibulskis, Stacey B Gabriel, David Altshuler, and Mark J Daly. A framework for variation discovery and genotyping using next-generation DNA sequencing data. *Nat Genet*, 43(5):491–498, May 2011.

[11] Jennifer L Freeman, George H Perry, Lars Feuk, Richard Redon, Steven A McCarroll, David M Altshuler, Hiroyuki Aburatani, Keith W Jones, Chris Tyler-Smith, Matthew E Hurles, Nigel P Carter, Stephen W Scherer, and Charles Lee. Copy number variation: New insights in genome diversity. *Genome Research*, 16(8):949–961, August 2006.

[12] Errol C. Friedberg. DNA damage and repair. *Nature*, 421(6921):436–440, January 2003.

[13] Anthony J.F. Griffiths. *Introduction to genetic analysis*. W.H. Freeman, New York, 2008.

[14] Stephen S. Hall. Journey to the genetic interior. *Scientific American*, 307(4):80–84, September 2012.

[15] Nils Homer. SourceForge.net: srma. http://sourceforge.net/apps/mediawiki/srma, January 2013.

[16] Nils Homer and Stanley F Nelson. Improved variant discovery through local re-alignment of short-read next-generation sequencing data using SRMA. *Genome Biology*, 11(10):R99, 2010.

[17] Adrienne Kitts and Stephen Sherry. The single nucleotide polymorphism database (dbSNP) of nucleotide sequence variation. In *The NCBI Handbook [Internet]*, page Chapter 5. Bethesda (MD): National Center for Biotechnology Information (US), February 2002.

[18] Daniel C Koboldt, Ken Chen, Todd Wylie, David E Larson, Michael D McLellan, Elaine R Mardis, George M Weinstock, Richard K Wilson, and Li Ding. VarScan: variant detection in massively parallel sequencing of individual and pooled samples. *Bioinformatics (Oxford, England)*, 25(17):2283–2285, September 2009. PMID: 19542151.

[19] Daniel C. Koboldt, Qunyuan Zhang, David E. Larson, Dong Shen, Michael D. McLellan, Ling Lin, Christopher A. Miller, Elaine R. Mardis, Li Ding, and Richard K. Wilson. VarScan 2: Somatic mutation and copy number alteration discovery in cancer by exome sequencing. *Genome Research*, 22(3):568–576, March 2012.

[20] Yuichi Kodama, Martin Shumway, and Rasko Leinonen. The sequence read archive: explosive growth of sequencing data. *Nucleic Acids Research*, 40(D1):D54–D56, January 2012. PMID: 22009675 PMCID: PMC3245110.

[21] Samuel Levy, Granger Sutton, Pauline C Ng, Lars Feuk, Aaron L Halpern, Brian P Walenz, Nelson Axelrod, Jiaqi Huang, Ewen F Kirkness, Gennady Denisov, Yuan Lin, Jeffrey R MacDonald, Andy Wing Chun Pang, Mary Shago, Timothy B Stockwell, Alexia Tsiamouri, Vineet Bafna, Vikas Bansal, Saul A Kravitz, Dana A Busam, Karen Y Beeson, Tina C McIntosh, Karin A Remington, Josep F Abril, John Gill, Jon Borman, Yu-Hui Rogers, Marvin E Frazier, Stephen W Scherer, Robert L Strausberg, and J Craig Venter. The diploid genome sequence of an individual human. *PLoS biology*, 5(10):e254, September 2007. PMID: 17803354.

[22] Heng Li. A statistical framework for SNP calling, mutation discovery, association mapping and population genetical parameter estimation from sequencing data. *Bioinformatics (Oxford, England)*, 27(21):2987–2993, November 2011. PMID: 21903627.

[23] Heng Li and Richard Durbin. Fast and accurate short read alignment with burrows-wheeler transform. *Bioinformatics (Oxford, England)*, 25(14):1754–1760, July 2009. PMID: 19451168.

[24] Heng Li and Richard Durbin. Fast and accurate long-read alignment with burrows-wheeler transform. *Bioinformatics*, 26(5):589–595, March 2010. PMID: 20080505 PMCID: PMC2828108.

[25] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, and Richard Durbin. The sequence Alignment/Map format and SAMtools. *Bioinformatics*, 25(16):2078–2079, August 2009. PMID: 19505943 PMCID: PMC2723002.

[26] Heng Li and Nils Homer. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in Bioinformatics*, 11(5):473–483, September 2010.

[27] Wentian Li. On parameters of the human genome. *Journal of theoretical biology*, 288:92–104, November 2011. PMID: 21821053.

[28] Lin Liu, Yinhu Li, Siliang Li, Ni Hu, Yimin He, Ray Pong, Danni Lin, Lihua Lu, and Maggie Law. Comparison of next-generation sequencing systems. *Journal of Biomedicine and Biotechnology*, 2012, 2012. PMID: 22829749 PMCID: PMC3398667.

[29] Kerensa E McElroy, Fabio Luciani, and Torsten Thomas. GemSIM: general, error-model based simulator of next-generation sequencing data. *BMC genomics*, 13:74, 2012. PMID: 22336055.

[30] Lucia Musumeci, Jonathan W Arthur, Florence SG Cheung, Ashraful Hoque, Scott Lippman, and Juergen KV Reichardt. Single nucleotide differences (SNDs) in the dbSNP database may lead to errors in genotyping and haplotyping studies. *Human mutation*, 31(1):67–73, January 2010. PMID: 19877174 PMCID: PMC2797835.

[31] Einar Nielsen, Jakob Hemmer Hansen, Peter Foged Larsen, and Dorte Bekkevold. Population genomics of marine fishes: identifying adaptive variation in space and time. *Molecular Ecology*, 18(15):3128–3150, August 2009.

[32] R Ophir and D Graur. Patterns and rates of indel evolution in processed pseudogenes from humans and murids. *Gene*, 205(1-2):191–202, December 1997.

[33] Konrad Paszkiewicz and David J Studholme. De novo assembly of short sequence reads. *Briefings in Bioinformatics*, 11(5):457–472, September 2010.

[34] Helen Pearson. Genetics: What is a gene? *Nature*, 441(7092):398–401, May 2006.

[35] Ji Qi, Fangqing Zhao, Anne Buboltz, and Stephan C Schuster. inGAP: an integrated next-generation genome analysis pipeline. *Bioinformatics (Oxford, England)*, 26(1):127–129, January 2010. PMID: 19880367.

[36] Torbjørn Rognes. Faster smith-waterman database searches with inter-sequence SIMD parallelisation. *BMC Bioinformatics*, 12:221, 2011.

[37] R Sachidanandam, D Weissman, S C Schmidt, J M Kakol, L D Stein, G Marth, S Sherry, J C Mullikin, B J Mortimore, D L Willey, S E Hunt, C G Cole, P C Coggill, C M Rice, Z Ning, J Rogers, D R Bentley, P Y Kwok, E R Mardis, R T Yeh, B Schultz, L Cook, R Davenport, M Dante, L Fulton, L Hillier, R H Waterston, J D McPherson, B Gilman, S Schaffner, W J Van Etten, D Reich, J Higgins, M J

Daly, B Blumenstiel, J Baldwin, N Stange-Thomann, M C Zody, L Linton, E S Lander, and D Altshuler. A map of human genome sequence variation containing 1.42 million single nucleotide polymorphisms. *Nature*, 409(6822):928–933, February 2001.

[38] L K Sengupta, Susmita Sengupta, and Munna Sarkar. Pharmacogenetic applications of the post genomic era. *Current Pharmaceutical Biotechnology*, 3(2):141–150, June 2002.

[39] T F Smith and M S Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, March 1981.

[40] Helga Thorvaldsdottir, James T. Robinson, and Jill P. Mesirov. Integrative genomics viewer (IGV): high-performance genomics data visualization and exploration. *Briefings in Bioinformatics*, April 2012.

[41] Hester M. Wain, Elspeth A. Bruford, Ruth C. Lovering, Michael J. Lush, Mathew W. Wright, and Sue Povey. Guidelines for human gene nomenclature. *Genomics*, 79(4):464–470, April 2002.

[42] David L. Wheeler, Tanya Barrett, Dennis A. Benson, Stephen H. Bryant, Kathi Canese, Vyacheslav Chetvernin, Deanna M. Church, Michael DiCuccio, Ron Edgar, Scott Federhen, Michael Feolo, Lewis Y. Geer, Wolfgang Helmberg, Yuri Kapustin, Oleg Khovayko, David Landsman, David J. Lipman, Thomas L. Madden, Donna R. Maglott, Vadim Miller, James Ostell, Kim D. Pruitt, Gregory D. Schuler, Martin Shumway, Edwin Sequeira, Steven T. Sherry, Karl Sirotkin, Alexandre Souvorov, Grigory Starchenko, Roman L. Tatusov, Tatiana A. Tatusova, Lukas Wagner, and Eugene Yaschenko. Database resources of the national center for biotechnology information. *Nucleic Acids Research*, 36(Database issue):D13–D21, January 2008. PMID: 18045790 PMCID: PMC2238880.

[43] Alec Wysoker, Kathleen Tibbetts, and Tim Fennell. Picard, December 2012.

[44] Ruibin Xi, Tae-Min Kim, and Peter J Park. Detecting structural variations in the human genome using next generation sequencing. *Briefings in Functional Genomics*, 9(5-6):405–415, December 2010.

9 Appendix

The appendix consists of two parts. First a section with program code from that was written as part of the thesis. Second an index of important terms that are used in the thesis.

9.1 Program files

9.1.1 Generate statistics from VCF file

```
#!/usr/bin/env python

# Generate statistics from VCF file. Namely the frequency of SNPs,
# deletions, insertions and MNPs, as well as the number of lines
# (excluding headers)

# If a line contains multiple variants, one is chosen at random! Thus
# the true number of variants is higher than the reported number of
# lines. Output is written to given file or, if not present, to
# standard out

# Usage:
# python generateVCFStatistics.py input.vcf output.stats


import sys,random

varfile=""
outputfile=""

def parseinput():
    try:
        global varfile,outputfile
        varfile = sys.argv[1]
        if (len(sys.argv)>2):
            outputfile = sys.argv[2]
    except:
        print "Usage: %s variantfile outputfile" % sys.argv[0]

def generateStats():
    varinput = open(varfile,'r')

    nSNP=0
    nMNP=0
    nDel=0
    nIns=0

    for line in varinput:
        if (line[0]=='#'):
            continue
        words = line.split("\t")
        ref = words[3]
        altwords = words[4].split(",")
        # draw a random variant, if there are more
        alt = altwords[random.randint(0,len(altwords)-1)]

        if (len(ref)==len(alt)):
            if (len(ref)>1):
                nMNP=nMNP+1
            else:
```

```
                nSNP=nSNP+1
        if (len(ref)>len(alt)):
            nDel=nDel+1
        if (len(ref)<len(alt)):
            nIns=nIns+1

    total = nSNP+nMNP+nDel+nIns+0.0
    printstr = "pSNP=%f\npMNP=%f\npdel=%f\npins=%f\ndbSNPsize=%f"
    if (outputfile==""):
        print printstr % (nSNP/total, nMNP/total, nDel/total, nIns/total
        , total)
    else:
        outfile = open(outputfile,"w")
        outfile.write(printstr % (nSNP/total, nMNP/total, nDel/total,
        nIns/total, total))
        outfile.close()

if __name__ == "__main__":
    parseinput()
    random.seed(314)
    generateStats()
```

9.1.2 Generate VCF files for testing

```
#!/usr/bin/env python

# Generate test VCF files.

# Input: vcf from variant database, for instance 1000G or dbSNP
# parameters: SNP and indel frequency (both homozygous and heterozygous)
# input file name and output file name prefix
# frequency Pu of unknown variants

# The algorithm will first open the variant file, and count the number
# of SNPs, MNPs, insertions and deletions. Then it will generate three
# files: outputprefix.varA.vcf outputprefix.varB.vcf and
# outputprefix.newdatabase.vcf.  Subsequently, it parse one and one
# line. First


import sys,operator,random

# Constants:
# rates in dbSNP:
pSNP = 0.84276
pdel = 0.08056
pins = 0.07264
pMNP = 0.00404
# number of variants in dbSNP
dbSNPsize = 52716087

# rates in sample human
sample_SNP = 0.78555
sample_hetero_SNP = 0.5485
sample_MNP = 0.01316
sample_hetero_MNP = 0.7243
sample_insert = 0.10216
sample_delete = 0.09912
sample_hetero_indel = 0.3205

sample_size = 4090620
```

```
38
39 downsampling_rate = 59505254./2857698560
40 sample_size = sample_size*downsampling_rate
41
42 novel_rate = 0.25
43
44 varfile=""
45 outputprefix=""
46
47 def parseinput():
48     try:
49         global varfile,outputprefix
50         varfile = sys.argv[1]
51         outputprefix = sys.argv[2]
52     except:
53         print "Usage: %s variantfile outputfile [statsfile]" % sys.argv
           [0]
54         sys.exit()
55     try:
56         var={}
57         execfile(sys.argv[3],var) # import from stats file
58         global pSNP,pMNP,pdel,pins,dbSNPsize
59         pSNP=var['pSNP']
60         pMNP=var['pMNP']
61         pdel=var['pdel']
62         pins=var['pins']
63         dbSNPsize=var['dbSNPsize']
64     except:
65         pass # use default statistics
66
67 def generateVariants():
68     varinput = open(varfile,'r')
69     outputfileA = open(outputprefix+".A.vcf",'w')
70     outputfileB = open(outputprefix+".B.vcf",'w')
71     outputfileDB = open(outputprefix+".DB.vcf",'w')
72
73     # current position (avoid overlaps)
74     chrA = 1
75     chrB = 1
76     Aindex = -1
77     Bindex = -1
78
79     for line in varinput:
80         # copy header
81         if (line[0]=='#'):
82             outputfileA.write(line)
83             outputfileB.write(line)
84             outputfileDB.write(line)
85             continue
86         words = line.split("\t")
87         ref = words[3]
88         altwords = words[4].split(",")
89         okA = 1
90         okB = 1
91         # check for overlap
92         if (chrA == words[0]):
93             if (int(words[1])<=Aindex):
94                 okA = 0
95         if (chrB == words[0]):
96             if (int(words[1])<=Bindex):
97                 okB = 0
```

```
98          if (len(words[3])>100 or len(words[4])>100):
99              # discard long variants
100             okA = okB = 0
101         # only use one variant from each line
102         alt = altwords[random.randint(0,len(altwords)-1)]
103         words[4] = alt
104         isused=0
105         # SNP
106         if (len(ref)==1 and len(alt)==1):
107             if (random.random()<(sample_size*sample_SNP/(dbSNPsize*pSNP)
                )):
108                 isused=1
109                 if (random.random()<sample_hetero_SNP):
110                     # heterozygous SNP, remove from one haplotype
111                     if (random.random()<0.5):
112                         okA=0
113                     else:
114                         okB=0
115         # MNP (may in reality be a SNP like (AT->AG))
116         elif (len(ref)==len(alt)):
117             if (random.random()<(sample_size*sample_MNP/(dbSNPsize*pMNP)
                )):
118                 isused=1
119                 if (random.random()<sample_hetero_MNP):
120                     if (random.random()<0.5):
121                         okA = 0
122                     else:
123                         okB = 0
124         # insertion (may be complex (AX -> AYY))
125         elif (len(ref)<len(alt)):
126             if (random.random()<(sample_size*sample_insert/(dbSNPsize*
                pins))):
127                 isused=1
128                 if (random.random()<sample_hetero_indel):
129                     if (random.random()<0.5):
130                         okA = 0
131                     else:
132                         okB = 0
133         # deletion (may be complex (AXX -> AY))
134         elif (len(ref)>len(alt)):
135             if (random.random()<(sample_size*sample_delete/(dbSNPsize*
                pdel))):
136                 isused=1
137                 if (random.random()<sample_hetero_indel):
138                     if (random.random()<0.5):
139                         okA = 0
140                     else:
141                         okB = 0
142
143         if (isused and okA):
144             outputfileA.write("\t".join(words))
145             chrA = words[0]
146             Aindex = int(words[1])+len(words[3])
147             okA=0
148         if (isused and okB):
149             outputfileB.write("\t".join(words))
150             chrB = words[0]
151             Bindex = int(words[1])+len(words[3])
152             okB=0
153         if (not isused or random.random()>novel_rate):
154             outputfileDB.write(line)
```

```
155     outputfileA.close()
156     outputfileB.close()
157     outputfileDB.close()
158
159 if __name__ == "__main__":
160     parseinput()
161     random.seed(314)
162     generateVariants()
```

9.1.3 Correct variant files

```
1  #!/usr/bin/env python
2  # Correct variant files
3  # Converts variants in one file to equivalent variants in another file
4
5  import sys,bisect
6
7  truthfile=""
8  variantfile=""
9  referencefile=""
10 outputfile=""
11
12 statsequal=0
13 statskept=0
14 statschangedfrom=0
15 statschangedto=0
16 statsignored=0
17 statsduplicate=0
18
19 # DEBUG = write all changes in the variant file?
20 DEBUG = False
21
22 if (len(sys.argv)<4):
23     print "Usage: %s truth_vcf input_vcf reference_sequence [output_vcf]
       [length]" % sys.argv[0]
24     sys.exit()
25 truthfile=open(sys.argv[1],"r")
26 variantfile=open(sys.argv[2],"r")
27 referencefile=open(sys.argv[3],"r")
28 try:
29     outputfile=open(sys.argv[4],"w")
30 except:
31     outputfile=sys.stdout
32
33 referencefile.readline()
34 reference = "".join([line.rstrip('\n') for line in referencefile])
35
36 # truth variants
37 truth=[]
38 # trial variants
39 variants=[]
40 # result variants after correction
41 newvariants=[]
42 # window size
43 wsize = 100
44 try:
45     wsize = int(sys.argv[5])
46 except:
47     pass
48
49 # read input
```

```
50  for line in truthfile:
51      if (line[0]=="#"):
52          # Re-use header file from truth file
53          outputfile.write(line)
54          continue
55      elems = line.split("\t")
56      alts = elems[4].split(",")
57      for alt in alts:
58          truth.append([int(elems[1]),elems[3],alt,len(elems[3])-len(alt)
            ])
59
60  for line in variantfile:
61      if (line[0]=="#"):
62          continue
63      elems = line.split("\t")
64      alts = elems[4].split(",")
65      for alt in alts:
66          variants.append([int(elems[1]),elems[3],alt,len(elems[3])-len(
            alt),float(elems[5])])
67
68  # write data lines in minimal VCF format
69  def toVCFformat(varlist):
70      if (not varlist):
71          return False
72      global wsize
73      res = ['1']
74      res.append(str(varlist[0]))
75      res.append(".")
76      res.append(varlist[1])
77      res.append(varlist[2])
78      res.append(str(varlist[4]))
79      res.append(".")
80      res.append(".\n")
81      return "\t".join(res)
82
83  # Subroutine: check if two subsets of truth file and variant file is
    equal.
84  # Returns true if equal, false otherwise. Can be time consuming
85  def checkequal(truthset,variantset):
86      global truth,variants,reference
87      start=min(truth[truthset[0]][0],variants[variantset[0]][0])
88      end=max(truth[truthset[-1]][0]+len(truth[truthset[-1]][1]),variants[
        variantset[-1]][0]+len(variants[variantset[-1]][1]))
89
90      # Check if either set is inconsistent (variants overlapping)
91      for i in range(1,len(variantset)):
92          if ((variants[variantset[i]][0]-variants[variantset[i-1]][0]-len
            (variants[variantset[i-1]][1]))<0):
93              return False
94      for i in range(1,len(truthset)):
95          if ((truth[truthset[i]][0]-truth[truthset[i-1]][0]-len(truth[
            truthset[i-1]][1]))<0):
96              return False
97
98      refstr = reference[start-1:end-1]
99      truthstr=list(refstr)
100     varstr=list(refstr)
101     # generate alternative haplotypes
102     try:
103         for var in variantset:
104             curvr = variants[var]
```

```
105             for a in range(1,len(curvr[1])):
106                 # remove excess reference nucleotides (deletion)
107                 varstr[curvr[0]-start+a]=''
108             if (varstr[curvr[0]-start]!=curvr[1][0]):
109                 # check that first nucleotide matches reference
110                 print "ERROR!",curvr,varstr,curvr[0],start
111             # insert alternative sequence
112             varstr[curvr[0]-start]=curvr[2]
113         for tr in truthset:
114             curtr = truth[tr]
115             for a in range(1,len(curtr[1])):
116                 truthstr[curtr[0]-start+a]=''
117             truthstr[curtr[0]-start]=curtr[2]
118     except:
119         return False
120     if ("".join(truthstr)=="".join(varstr)):
121         return True
122     return False
123
124 # 0 -> no match
125 # 1 -> identical match
126 # [truth set,variant set] -> equivalent, but not identical match
127 def getmatch(varidx,var,curwsize=0):
128     if (curwsize == 0):
129         curwsize = wsize
130     pos = var[0]
131     # loop through all truths at same position and look for truth
        variants
132     # identical to the current variant.
133     # in most cases 0 or very few iterations due to binary search.
134     curt = bisect.bisect_left(truth,[pos,"0","0"])
135     while (curt != len(truth)):
136         if (truth[curt][0]!=pos):
137             break
138         if (truth[curt][2] == var[2] and truth[curt][1] == var[1]):
139             return 1
140         curt = curt+1
141
142     # find leftmost/rightmost variant in both windows
143     curtleft = bisect.bisect_left(truth,[pos-curwsize/2,"0","0"])
144     curtright = bisect.bisect_left(truth,[pos+3*curwsize/2,"z","z"])
145     curvarright = bisect.bisect_left(variants,[pos+curwsize,"z","z"])
146     # if 0 variants in truth: no match
147     if (curtleft == curtright):
148         return 0
149
150     curtw = curtright-curtleft
151     varw = curvarright-varidx-1
152     total = curtw+varw
153
154     # Limit window if there are too many variants, warn if there are
        many
155     if (total>25):
156         print "error, %d variants, more than 25, cut window in half to %
            d" % (total,int(curwsize/2))
157         return getmatch(varidx,var,int(curwsize/2))
158     if (total>16):
159         print "warning, %d variants" % total
160
161     # create both haplotypes and compare
162     for i in range(1,1<<curtw):
```

```
        change=0
        for a in range(curtw):
            if (i&(1<<a)):
                change = change + truth[curtleft+a][3]
        for j in range(0,1<<varw):
            change2 = variants[varidx][3]
            for a in range(varw):
                if (j&(1<<a)):
                    change2 = change2 + variants[varidx+a+1][3]
            # Fast check: if length of haplotypes is different, continue
            if (change!=change2):
                continue

            truthset=[]
            variantset=[varidx]
            for a in range(curtw):
                if (i&(1<<a)):
                    truthset.append(curtleft+a)
            for a in range(varw):
                if (j&(1<<a)):
                    variantset.append(varidx+a+1)
            # thorough test
            if (checkequal(truthset,variantset)):
                return [truthset,variantset]
    return 0

# Already matched variants that should not be counted as false
# positives, but may be true positives in a diploid genome.
ignoredvariants = []

# go through all variants in trial file, and try to match
for varidx,var in enumerate(variants):
    # output progress
    if (varidx/(len(variants)/10)!=((varidx+1)/(len(variants)/10))):
        print "progress: %d/10" % (1+varidx/(len(variants)/10))
    result = getmatch(varidx,var)
    if (result == 0):
        if varidx in ignoredvariants:
            ignoredvariants.remove(varidx)
            statsignored = statsignored + 1
            continue
        newvariants.append(var)
        statskept = statskept + 1
        continue
    if (result == 1):
        if (varidx in ignoredvariants):
            ignoredvariants.remove(varidx)
        newvariants.append(var)
        statsequal = statsequal + 1
        continue
    statschangedto = statschangedto + len(result[0])
    statschangedfrom = statschangedfrom + len(result[1])
    statsignored = statsignored + 1
    varscore = 10000
    for res in result[1]:
        varscore = min(varscore,variants[res][-1])
    if (DEBUG):
        begin = min(truth[result[0][0]][0],variants[result[1][0]][0])
        end = max(truth[result[0][-1]][0],variants[result[1][-1]][0])
        print "Reference:",begin,reference[begin-1:end],end
        print "Change to:"
```

```
224
225     for trval in result[0]:
226         newvariants.append(truth[trval]+[varscore])
227         if (DEBUG):
228             print truth[trval][0:3]
229     if (DEBUG):
230         print "Change from:"
231
232     for varval in result[1]:
233         if (varval!=varidx):
234             ignoredvariants.append(varval)
235         if (DEBUG):
236             print variants[varval][0:3]
237
238 newvariants.sort()
239 # if multiple variants at same location, join to one comma-separated
    entry
240 for i in range(1,len(newvariants)):
241     if (newvariants[i][1]==newvariants[i-1][1]):
242         if (newvariants[i][0]==newvariants[i-1][0]):
243             if (newvariants[i][2]==newvariants[i-1][2]):
244                 newvariants[i-1]=False
245                 statsduplicate = statsduplicate +1
246                 continue
247             newvariants[i][2]=newvariants[i-1][2]+","+newvariants[i][2]
248             newvariants[i-1]=False
249             continue
250
251
252 for var in newvariants:
253     res = toVCFformat(var)
254     if (res):
255         outputfile.write(res)
256
257 outputfile.close()
258 print "Statistics:"
259 print "Variants equal to reference: ",statsequal
260 print "Variants wrong, kept as is: ",statskept
261 print "Wrong variants removed: ",statsignored
262 print "duplicates removed: ",statsduplicate
263 print "Variants changed: ",statschangedfrom
264 print "Variants changed into: ",statschangedto
```

9.1.4 Split variant file according to depth

```
1  #!/usr/bin/env python
2
3  # Split a given variant file according to variant depth. Use a (kind
4  # of) running average instead of reported depth
5  #
6  # Usage:
7  # python variantdepth.py depthfile variants.vcf [interval width]
8  #
9  # will create files variant.[interval].vcf, where [interval] is the
10 # upper limit of depth that this file contains
11 #
12 # Header files are kept as is.
13
14 import sys,os
15 varfile=""
16
```

```
def filtervars():
    try:
        global depthfile,varfile,intervalsize
        depthfile = sys.argv[1]
        varfile = sys.argv[2]
        intervalsize = 1
        if (len(sys.argv)>3):
            intervalsize = int(sys.argv[3])
    except:
        print "Usage: %s depthfile varfile [interval width] " % sys.argv
        [0]
        sys.exit()

    depths = open(depthfile,"r")
    variants = open(varfile,"r")
    (root,ext) = os.path.splitext(varfile)

    maxdepth=0
    depthdict = {}
    for line in depths:
        elems = line.split()
        depthdict[int(elems[1])]=int(elems[2])
        maxdepth = max(maxdepth,int(elems[2]))

    numintervals=1+(maxdepth+1)/intervalsize
    filedict = {}
    for i in range(numintervals):
        filedict[i] = open(root+"."+str(i*intervalsize)+".vcf","w")
    intro = ""
    introdone = False
    for line in variants:
        if (line[0]=="#"):
            intro = intro + line
            continue
        else:
            if (not introdone):
                for i in range(numintervals):
                    filedict[i].write(intro)
                introdone = True
        elems = line.split()
        depthleft = 0
        depthright = 0
        margin = 21 # window size
        curpos = int(elems[1])-margin/2

        for i in range(margin+1):
            if (curpos in depthdict):
                if (curpos<=int(elems[1])):
                    depthleft = max(depthleft,depthdict[curpos])
                if (curpos>=int(elems[1])):
                    depthright = max(depthright,depthdict[curpos])
            curpos = curpos + 1
        depth = min(depthleft,depthright)

        filedict[(depth+intervalsize-1)/intervalsize].write(line)
    for i in range(numintervals):
        filedict[i].close()

if __name__ == "__main__":
    filtervars()
```

9.1.5 Calculate data for sensitivity/precision graph

```
#!/usr/bin/env python

# Annotates a variant file with "correct" or "false" in the ID
# column. Sorts according to quality. Removes header. Split in two
# files, one for SNPs and one for indels. Rest are discarded.

# Needs truth vcf file and

import sys,bisect

truthfile=""
variantfile=""
outputfilesnp=""
outputfileindel=""

if (len(sys.argv)<4):
    print "Usage: %s truth_vcf input_vcf  output_vcf_prefix" % sys.argv
    [0]
    sys.exit()
truthfile=open(sys.argv[1],"r")
variantfile=open(sys.argv[2],"r")

outputfilesnp=open(sys.argv[3]+".snp_statistics","w")
outputfileindel=open(sys.argv[3]+".indel_statistics","w")

# truth variants
truthsnp={}
truthindel={}
# trial variants
variantsnp=[]
variantindel=[]

# read input
for line in truthfile:
    if (line[0]=="#"):
        continue
    elems = line.split("\t")
    alts = elems[4].split(",")
    for alt in alts:
        st = elems[1]+"-"+elems[3]+"-"+alt
        if (len(elems[3])==1 and len(alt)==1):
            truthsnp[st]=1
        elif(len(elems[3])==1 or len(alt)==1):
            truthindel[st]=1

for line in variantfile:
    if (line[0]=="#"):
        continue
    elems = line.split("\t")
    alts = elems[4].split(",")
    elems[0]= "%09.2f" % float(elems[5])
    for alt in alts:
        st = elems[1]+"-"+elems[3]+"-"+alt
        if (len(elems[3])==1 and len(alt)==1):
            elems[4]=alt
            if st in truthsnp:
                elems[2]="correct"
            else:
                elems[2]="false"
```

```
59              variantsnp.append("\t".join(elems))
60          elif (len(elems[3])==1 or len(alt)==1):
61              elems[4]=alt
62              if st in truthindel:
63                  elems[2]="correct"
64              else:
65                  elems[2]="false"
66              variantindel.append("\t".join(elems))
67
68  variantsnp.sort()
69  variantindel.sort()
70
71  numSNPs = len(truthsnp)
72  print "number␣of␣SNPs:␣",numSNPs
73  numcorrect=0.
74  numfalse=0.
75  for line in reversed(variantsnp):
76      # output raw SNPs
77      #    outputfilesnp.write(line)
78      elems = line.split("\t")
79      if (elems[2]=='correct'):
80          numcorrect = numcorrect+1
81      else:
82          numfalse = numfalse+1
83      # write current sensitivity and precision
84      outputfilesnp.write("%.3f␣%.3f\n" % ((100*numcorrect/numSNPs),(100*
        numcorrect/(numcorrect+numfalse))))
85  outputfilesnp.close()
86
87  numindels = len(truthindel)
88  print "number␣of␣indels:␣",numindels
89  numcorrect=0.
90  numfalse=0.
91  for line in reversed(variantindel):
92      # output raw indels
93      #    outputfileindel.write(line)
94      elems = line.split("\t")
95      if (elems[2]=='correct'):
96          numcorrect = numcorrect+1
97      else:
98          numfalse = numfalse+1
99      outputfileindel.write("%.3f␣%.3f\n" % ((100*numcorrect/numindels)
        ,(100*numcorrect/(numcorrect+numfalse))))
100 outputfileindel.close()
```

Index